SICS Technical Report
T91:06

ISRN : SICS-T—91/06-SE
ISSN : 1100-3154

A Parser for GDL written in Sicstus Prolog

by

Martin Aronsson

July 1991

Swedish Institute of Computer Science
Box 1263, S-164 29 KISTA, SWEDEN

A Parser for GDL written in Sicstus Prolog

Martin Aronsson
Swedish Institute of Computer Science
Box 1263
S-164 28 Kista
Sweden

91-07-17

Abstract

A parser for a language called GDL (Graphical Description Language, a Basic-like language) is presented. GDL is a language used in the CAD system ArchiCAD®, which is a CAD system used by architects for drawing buildings. GDL is the language used by ArchiCAD® for plotting 3D views. The parser described here has been implemented in a project for planning the erection of buildings, where the GDL programs are used to get the coordinates and the different materials of different parts of the building. The report describes the general, basic plotting features of GDL and does not go into details about the more architectual features of GDL and ArchiCAD®. The parser is presented in detail, and an interpreter that runs the parsed code is also presented. The report contains three appendixes, which present GDL and the programs implementing the parser and the interpreter.

1 Introduction

The aim of this report is to describe how Sicstus Prolog can be used, and has been used, for implementing a parser and to show different implementation techniques in Sicstus Prolog. This will be done by parsing a Basic-like language used in ArchiCAD®, a CAD system for architectual drawings. The language is called GDL, Geometric Description Language. In ArchiCAD® the GDL files are used as macros for drawing different objects from a library into the three dimensional view of a building. Typical objects are different furnitures, installations etc, up to walls, slabs and entire flats. We will show how common statements in GDL are parsed, such as flow statements, expressions, input and output primitives, and some plot primitives. We will not show how to parse wall and slab statements, since these are highly specialized, and do not demonstrate any programming features that are not presented by simpler constructs. Beside that, the parser is actually used in a prototypical planning system for the construction site.

We have used DCG, Definite Clause Grammar, to implement most of the parser. The parser has three passes; the first pass reads all the letters in the file, the second pass forms symbols and tokens from the letters, and the third pass forms statements from the tokens from pass 2. Some error checking are performed, but we have not tried to implement an "intelligent" error routine.

After the file is parsed, the statements should be interpreted. Here the interpreter in this paper and the interpreter in the planning system differs. In the planning system we are interested in getting the coordinates of the different parts of the building, while in this paper the GM package of Sicstus Prolog has been used to plot the figures described by the parsed file. By this the user of this system can easily see how the GDL files are interpreted.

The paper first gives a brief description of how DCG, Definite Clause Grammar, is used for parsing. Then a short introduction to GDL is given. Section 4 presents the parser in detail, and section 5 presents the interpreter, which plots the figures. A small example is shown in section 6.

Appendix A presents the GDL language in detail, appendix B contains the code for the parser, and appendix C contains the code for the interpreter.

The work presented herein was done within the project Industrial Sicstus Prolog, and is intended as a programming example. Therefore the presentation is of a describing nature.

2 Description of DCG

DCG, Definite Clause Grammar, is a generalization of context-free grammars that are executable. A context-free grammar consist of a set of rules of the form

```
nonterminal --> body
```

where `nonterminal` is a nonterminal symbol and `body` is a sequence of one or more items, separated by commas. Each item is either a nonterminal symbol, or a sequence of terminal symbols.

A DCG consists of terminals, which are written inside lists ([terminal$_1$, ..., terminal$_n$]), rules consisting of a nonterminal item and a body of one or more items (i.e. terminals or nonterminals). An example of a DCG is

```
sent --> a,b.
a --> [f].
b --> [g].
```

Sicstus Prolog implements DCGs by translating them into ordinary Sicstus Prolog clauses. The translation adds two arguments to the end of the nonterminal items. These arguments are used to feed the grammar with a stream of symbols. This stream is the sentence that should be checked by the grammar. For example, to the grammar above, the query

```
sent([f,g],[])
```

succeeds, while

```
sent([f,h],[])
```

fails.

To look a bit more in detail what the translation does, the example above is in principle translated to the Sicstus Prolog program

```
sent(A, C) :- a(A, B), b(B, C).
a([f|R],R).
b([g|R],R).
```

Note how the arguments are used to pass the result from `a` to `b` for the query `sent([f,g],[])`.

A common technique is to add an argument to the DCG rules, which contains the internal structure that we are constructing from the sentence parsed, for example the compiled code. If we add such an argument to the example above, for example

```
sent(s(X,Y)) --> a(X),b(Y).
a(f) --> [f].
b(g) --> [g].
```

the corresponding Sicstus Prolog program looks like

```
sent(s(X,Y),A,C) :- a(X,A,B), b(Y,B,C).
a(f,[f|R],R).
b(g,[g|R],R).
```

and the query

```
sent(S,[f,g],[])
```

binds the variable `S` to `s(f,g)`.

Now suppose that we add a rule to the program above, namely

```
a(h) --> [h].
```

resulting in the Sicstus Prolog program

```
sent(s(X,Y),A,C) :- a(X,A,B), b(Y,B,C).
a(f,[f|R],R).
a(h,[h|R],R).
b(g,[g|R],R).
```

Also assume that some routine `change(S, S1)` is defined, which changes `S` to `S1`. For example, assume that `change` changes `s(f,g)` to `s(f,h)`. Then the query

```
sent(S,[f,g],[]), change(S,S1), sent(S1,Sentence,[]).
```

binds `Sentence` to `[f,h]`. So it is easy to use the grammar both to check the syntax of a sentence and to generate sentences.

A nice way of writing termination symbols is by using strings. When a file is read in letter by letter, we have to parse the ascii numbers instead of the letters. If we use the string markers, it is easy to see what we are parsing in the rules. For example,

```
comma --> [44].
```

and

```
comma --> ",".
```

denotes the same thing, but the last one is easier to read.

We have only scratched the surface of what can be done with DCGs. For a more complete description of what DCGs can be used for and how they are implemented, the reader is referred to, in principle, any introductory book presenting Prolog, for example [Ste86]. The Sicstus Prolog manual also presents DCG in some detail.

3 GDL

GDL (Geometric Description Language) is a three-dimensional, general-purpose geometric description language. The structure and syntax of a GDL program are similar to the BASIC language. A GDL program consists of statements which define geometric elements in the three-dimensional space, manipulate numeric data and character strings, handle variables and control the flow of the execution.

GDL contains 2D and 3D geometric basic elements which can serve as "bricks" to build up complex shapes.

The essential task of a GDL program is to describe a system of geometric objects, so the main "building bricks" of GDL are the geometric objects. There are two kinds of them: primitive and compound.

In order to minimize the number of parameters of geometric objects, each compound object is specified in a local coordinate system fixed to the object. For instance, a cylinder is defined by two data items (length and radius). It begins in the origin and is coaxial with the z-axis. In a description like this, the position of the objects can be defined by preceding coordinate transformations of the local coordinate system.

The geometric parameters of each body refer to the actual local coordinate system.

A small GDL program which draws a dishwasher is shown below.

```
! Dishwasher   19.08.1989.
!internal macro: -

pars a = 2',b = 2',c = 3'
mulx a/2'
muly b/2'

prism 4,c,0,0,2',0,2',2',0,2'

rotx 90

lin_ 0,3 1/2",0,2',3 1/2",0
```

```
lin_ 0,c-5",0,2',c-5",0
poly 4,1",c-4",23",c-4",23",c-1",1",c-1"

poly 4,0,c-10",6",c-10",6",c-7",0,c-7"

del top
end
```

Appendix A lists the syntax and the statements of the subpart of GDL that the parser recognizes.

4 The parser

The parser works in three passes. The first opens the file, reads all the letters in the file, and closes the file. The second pass forms symbols, numbers and other tokens from the letters. Pass three forms statements from the tokens from pass two.

4.1 Pass 1: Reading in the GDL file

Pass one is simple, a list is created which contains all the letters in the file. The letters are represented by their ascii numbers. The routine is a recursive predicate:

```
read_to_eof(S,X) :-
      get0(S,Z),
      (Z = -1 -> X = [eof] ; read_to_eof(S,R),X = [Z|R]).
```

where S is the stream to the file.

4.2 Pass 2: Parsing the tokens

Pass two is more complicated. Here symbols, numbers, and other tokens are formed from the letters. The top level rules are shown below.

```
parse_tokens([eof]) --> [eof],!.
parse_tokens(S) --> [32],!,parse_tokens(S).     % skip spaces
parse_tokens(S) --> [9],!,parse_tokens(S).      % skip tabs
parse_tokens(S) --> [33],!,parse_comment,       % skip comments
      parse_tokens(S).
parse_tokens([F|R]) --> token(F),!,parse_tokens(R).
parse_tokens(S) --> parse_error_to_eol,parse_tokens(S).
```

As can be seen extra spaces, tabs etc are skipped here, and will not bother us later. The only nonterminal that adds something to our internal structure is `token`.

```
token(label(X)) --> label(X).
token(num(X)) --> number(X).
token(string(X)) --> string(X).
token(word(X)) --> word(X).
token(X) --> special_symbol(X).
```

A token is a label, a number, a string, a word (sequence of characters) or a special symbol (for example '(', ')', +, ^, * etc). All these are simple to parse, except for numbers, which will be described in more detail.

According to the description of GDL, a number should match

{[+/-]integer_part[.]}
{[+/-][integer_part].fractional_part} [E[+/-]exponent]

where integer_part, fractional_part and exponent are sequences of digits. Examples of numbers are: 1.0, -5.5, 6, 9E25, .25, -.25

In addition to this, the numbers can have (where appropriate) feet and/or inches as units instead of meters, which is the default, and also be rational numbers. We have choosen to have all numbers in metres, so we transform feet and inches into metres.

All numbers are rounded to 6 digits precision, after they have been parsed and formed, which is done by num.

```
number(N1) --> num(N),{roundoff(N,6,N1)}.
```

A number may or may not contain an integer part. If it contains an integer part, it is parsed and we continue to see if it has a fractional part. If there is no integer part, we try to find a fractional part.

```
num(N) --> sequence_of_digits(X),!,num11(N,X).% The integer part
num(N) --> num1(N,0).                          % No integer part
```

num11 is used when the integer part has been parsed. If there is a fractional or relational part, the first rule of num11 succeeds, otherwise the number is an integer and the second rule is applicable. unit takes care of transforming the number into metres, if the unit is feet or inches.

```
num11(N,X) --> num1(N,X),!.          % rational or fractional part
num11(N,X) -->
      {name(X1,X)},unit(N,X1).       % Just integer part
```

num1 takes care of rational and fractional parts. First it tries to parse a rational number, and if that does not succeed, it tries to parse a fraction and a possible exponent. Units must be taken care of here as well.

```
num1(N,X) -->                                  % Rational
      spaces,sequence_of_digits(T),!,"/",sequence_of_digits(B),
      {name(T1,T),name(B1,B),name(X1,X),N1 is X1 + (T1 / B1)},
      unit(N,N1).
num1(N,X) -->                                  % fractional part
      ".",!,sequence_of_digits(F),             % with exponent
      exp(N1,X,F),unit(N,N1).
```

The unit conversion is easy. Since the parsed number is sent in the second argument of the unit call, all that has to be done is to parse the unit, if there is one, and do the transformation.

```
unit(N,N1) --> [39],!,{N is N1 * 0.31}.        % ' (foot)
unit(N,N1) --> [34],!,{N is N1 * 0.025}.       % " (inch)
unit(N,N) --> [].
```

An exponential is parsed roughly by parsing a sequence of digits.

4.3 Pass 3: Parsing statements

The third pass forms statements from the items from the second pass. The only thing that cause trouble here is expressions. All other things can be parsed from left to right, but expressions can have operators of different precedence, which means that an expression tree must be built up.

The main predicate is parse_statement, which is called with the result of the second pass.

```
parse_statements([exit]) --> [eof],!.      % We are finished
parse_statements(S) -->                    % skip rows beginning
      [lf],!,parse_statements(S).          % with a linefeed
parse_statements([label(X)|R]) -->         % labels
      [label(X)],!,parse_statements(R).
parse_statements([directive(F)|R]) -->     % directives
      directive(F),!,parse_statements(R).
parse_statements([flow(F)|R]) -->          % flow statements
      flow(F),!,parse_statements(R).
parse_statements([io(F)|R]) -->            % input-output statem.
      io(F),!,parse_statements(R).
parse_statements([trans(F)|R]) -->         % graphical transform.
      transformation(F),!,parse_statements(R).
parse_statements([two_dim(F)|R]) -->       % two dimensional
      two_dim(F),!,parse_statements(R).    % statements
parse_statements([three_dim(F)|R]) -->     % three dimensional
      three_dim(F),!,parse_statements(R).  % statements
parse_statements(S) -->                    % parsing error, parse
      parse_error,parse_statements(S).     % error and continue
```

The statements are parsed by their "class-rule". For example, a directive statement is parsed by one of the directive rules:

```
directive(let(Varnam,E)) -->
      [word(let),word(Varnam),'='],expression(E),[lf].
directive(model(M)) --> [word(model)],!,model(M),[lf].
directive(pen(E)) --> [word(pen)],expression(E),[lf].
directive(box(Xmin,Xmax,Ymin,Ymax,Zmin,Zmax)) --> [word(box)],
      expression(Xmin),[','],optional_lf,
      expression(Xmax),[','],optional_lf,
      expression(Ymin),[','],optional_lf,
      expression(Ymax),[','],optional_lf,
      expression(Zmin),[','],optional_lf,
      expression(Zmax),[lf].
directive(box(cont)) --> [word(box),word(cont)],[lf].
directive(box(pause)) --> [word(box),word(pause)],[lf].
directive(radius(Rmin,Rmax)) --> [word(radius)],
      expression(Rmin),[','],optional_lf,expression(Rmax),[lf].
directive(sfline(N)) --> [word(sfline)],expression(N),[lf].
directive(base) --> [word(base),lf].
```

The typing of the items from pass two (for example `word(Varnam)`) makes it easier to parse the statements. As before, the argument to the rules is the structure that is built up and returned.

All other sets of statements are parsed analogously.

To parse a loop, (for-next loops), we have choosen to put the code inside the structure of the loop statement:

```
flow(for(let(Varnam,StVal),SlVal,1.0,Stmnts)) --> [word(for)],
      [word(Varnam),'='],expression(StVal),
      [word(to)], expression(SlVal),[lf],
      parse_statements_in_loop(SlVal,1.0,Stmnts).
flow(for(let(Varnam,StVal),SlVal,StepVal,Stmnts)) -->
      [word(for)],
      [word(Varnam),'='],expression(StVal),
      [word(to)], expression(SlVal),
      [word(step)], expression(StepVal), [lf],
      parse_statements_in_loop(SlVal,StepVal,Stmnts).
```

When the `for`-statement is parsed, the statements until the corresponding `next`-statement is parsed in the rule `parse_statements_in_loop`. This rule is analogous to `parse_statements`, except that when a next-statement is reached, the `SlVal` and `StepVal` are incorporated in that statement, and `parse_statements_in_loop` terminates.

```
...
parse_statements_in_loop(SlVal,StVal,
                         [flow(next(V,StVal,SlVal))]) -->
      [word(next),word(V),lf],!.
...
```

Of course we could have been using `parse_statements` to parse statements in a loop also, by keeping track of when we are at the top level, in which case it should terminate with an `exit`-statement (`end`-statement), and when we are in a loop, in which case it should terminate with a `next`-statement.

Parsing expressions gives rise to some problems, as they can have operators of different precedence. We have divided the parsing of expressions into two (sub)passes. First one pass which parses the sub-expressions into operator-term couples. For example, 1 + 2 * 3 is parsed into the couples *c(1,+), c(2,*), end(3)*. These couples are then combined by another pass into an expression tree, which takes operator precedings etc into account. In case of equal precedence, the sub-expressions shall be evaluated from left to right, i.e `x / y / z` is the same as `(x / y) / z`.

```
expression(X) --> parse_couples(X1),
      {combine(X,X1,[])}.
```

`X1` holds the list of couples, which are passed to `combine`. `combine` is a DCG call, and will return the expression tree in `X`.

`parse_couple` is a recursive rule, and when it cannot find a term followed by an operator, that term is the end term of the current expression, and the recursion stops.

```
parse_couples([c(Term,Oper,N)|R]) -->
      simple_term(Term),operator(N,Oper),!,
      parse_couples(R).
parse_couples([end(Term)]) --> simple_term(Term).
```

`simple_term` takes care of all terms that can be parsed from left to right, i.e. function calls, numbers etc.

```
simple_term(Z) --> [num(X),num(Y)],      % rational number
      {Z is X + Y}.
simple_term(X) --> [num(X)].
simple_term(X) --> ['+',num(X)].
simple_term(Y) --> ['-',num(X)],{Y is -X}.
simple_term(X) --> ['('],!,expression(X),[')'].
simple_term(constant(X)) --> constant(X).
simple_term(var(X)) --> variable(X).
simple_term(function(Name,Arg)) -->
      function(Name),['('],expression(Arg),[')'].
```

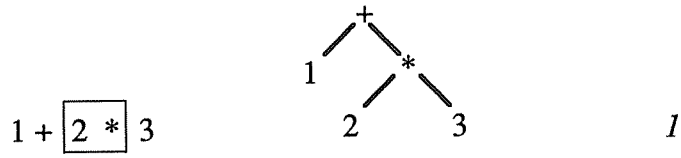
`combine` uses `combine1` for traversing the pairs. The first argument to `combine1` is an accumulating term, `RT` is to be instantiated depending on the next operator's precedence value, which the routine `insert` takes care of. In this way the expression tree is built up, and returned in the third argument when the end term is reached.

```
combine(Answ) --> [end(Answ)],!.                    % Just a number
combine(Answ) --> [c(Term,Oper,N)],
      combine1(operator(N,Oper,Term,RT),RT,Answ).

combine1(Sofar,Term,Sofar) --> [end(Term)].
combine1(Sofar,RT,Answ) --> [c(Term1,Oper,N1)],
      {insert(c(Term1,Oper,N1),Sofar,NSofar,RT,NRT)},
       combine1(NSofar,NRT,Answ).
```

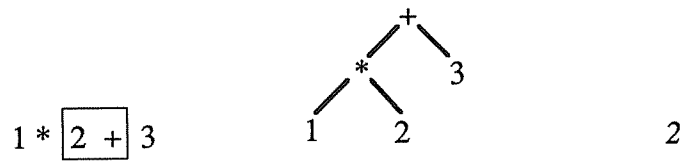
`insert` traverses the expression tree built sofar, residing in its second argument, and inserts the new term-operator pair in the tree according to the precedence value of the operator. As long as the new couple's precedence is lower than the considered node in the tree, the third clause is applicable and it continues to traverse the tree down the right branch. If the new couple's precedence value is equal to or higher than the current node in the tree, the couple is inserted above the current node, which becomes the new node's left branch. This is accomplished by the second clause. The first clause is used when the current node is the rightmost leaf, which is always a variable during the insertion. The new couple is inserted here, and the new node is returned.
The fourth and fifth arguments are used to unify and pass the right most leaf in the tree. They are used in the first and second clauses. In the first clause the leaf is changed, while in the second clause the old leaf is instantiated to the couple's term, and the new right-most leaf is returned in the fifth argument.

```
insert(c(Term1,Oper,N1),Var,operator(N1,Oper,Term1,Var1),
       Var,Var1) :-
      var(Var),!.
insert(c(Term1,Oper1,N1),operator(N,Oper,T1,T2),
       operator(N1,Oper1,operator(N,Oper,T1,T2),Var),
       Term1,Var) :-
      N =< N1,!.
insert(c(Term1,Op1,N1),operator(N,Op,T1,T2),
       operator(N,Op,T1,T2p),RT,NRT) :-
      N > N1,!,
      insert(c(Term1,Op1,N1),T2,T2p,RT,NRT).
```

Below are two expressions, their trees, and the number of the used `insert` clause (in italic). We have also marked the couple which is inserted by the `insert` clauses.



1



2

5 Running the processed GDL code

We have implemented an interpreter for the parsed GDL files, which plots the figures. The same main predicate is used in the planning system for the construction site, but there we are more interested in transforming the different statements into the object's coordinates, which are then written to a special file which the planning system then reads. Here the interpreter performs the same actions as in the ArchiCAD® system, i.e. plotting the objects and lines into a three-dimensional space. There are three views that reflects the

global coordinate system, and these are XY, XZ and YZ. We have used Sicstus GM library to plot the different lines and compound elements.

We have not implemented full GDL, and we are not even able to handle all constructs that the parser handles. This is because we do not need all constructs in the planning system, and also because it is fairly easy to add the statements that we have not implemented to the interpreter, so it should be easy for anybody else. The aim was to show how this system could be implemented, not to give a complete GDL implementation.

The structure of the interpreter is a common one: one main, recursive predicate, which gets the current statement to be executed. This predicate calls different other predicates depending on the current statement. In our case the main predicate has 5 arguments. The first argument is the Program Counter (PC), which is not implemented as a number but as a list of the statements which are to be executed, and the first element in that list is the statement to be executed. It will in fact inside the Sicstus Prolog implementation be realized as a pointer into the list consisting of all the statements in the program (c.f. a number into an array). The first argument could also be seen as a stream, where the first element in the stream is the statement to be executed.
The second argument is the whole program, and is used when a flow statement jumps somewhere to a label. This argument will refer to the same list as the first argument inside the Sicstus Prolog system.
The third argument is the variable environment, which is implemented as an assoc-list a'la Lisp, i.e. it is a list consisting of variable-value pairs. The fourth argument is the graphical states, a list of states where the first element is the current state. This representation makes it easy to create and return to different states. The states are represented by a matrix in the common way, i.e. 4 x 4 matrix, where the positions represent the following:

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ Dx & Dy & Dz & S \end{bmatrix}$$

where r_{ij} are used for rotations, Dx, Dy, Dz are used for transformations along the corresponding axis, and S is used for overall scaling. For further information about this see a book on computer graphics, for example [Fol84].

The matrix above is represented as a list of lists: [[r_{11}, r_{12}, r_{13}, 0], ..., [Dx, Dy, Dz, S]].

The fifth argument is a stack, which is used for gosub-calls. When a `gosub` statement is executed, the return address (i.e. the list of the rest of the statements in the first argument) is pushed on the stack, and when the `return` statement is reached, the topmost continuation is popped off and restored in the first argument.

For each class of statements, there is a special predicate which executes the statement. The appropriate arguments (program, environment etc) are passed to these predicates, and the changed structures are passed back to the main predicate.

Below are some of the clauses from the main predicate, together with some of the class-predicates.

```
inter([flow(X)|R],Pr,E,St,Stack) :- !,  % A flow stmnt jumps
      flow_int(X,R,E,Pr,St,Stack).      % so our recursion stops
inter([io(F)|R],Prog,Env,State,Stack) :- !,
      io_int(F,Env,Newenv),
      inter(R,Prog,Newenv,State,Stack).
inter([trans(F)|R],Prog,Env,State,Stack) :- !,
      trans_int(F,Env,State,Newstate),
      inter(R,Prog,Env,Newstate,Stack).
inter([directive(F)|R],Prog,Env,State,Stack) :- !,
      directive_int(F,Env,Env1),
      inter(R,Prog,Env1,State,Stack).

trans_int(addz(Expr),Env,[FS|States],[NewState,FS|States]) :- !,
      evaluate(Expr,E,Env),
      FS = [[A11,A12,A13,A14],[A21,A22,A23,A24],
            [A31,A32,A33,A34],[A41,A42,A43,A44]],
      A43p is A43 + E * A33,
      NewState = [[A11,A12,A13,A14],[A21,A22,A23,A24],
                  [A31,A32,A33,A34],[A41,A42,A43p,A44]].

directive_int(let(V,Expr),Env,Env1) :- !,
      evaluate(Expr,E,Env),
      make_env(V,E,Env,Env1).

flow_int(if_gosub(Expr1,Expr2),R,Env,Prog,State,Stack) :- !,
      evaluate(Expr1,Expr11,Env),
      (1 is integer(Expr11) ->                  % if-part true?
       evaluate(Expr2,Expr21,Env),
       Expr is integer(Expr21),
       append(_,[label(Expr)|Cont],Prog),       % find label
       inter(Cont,Prog,Env,State,[R|Stack]);    % return to cont
inter(R,Prog,Env,State,Stack)).                 % otherwise go on
```

6 An example

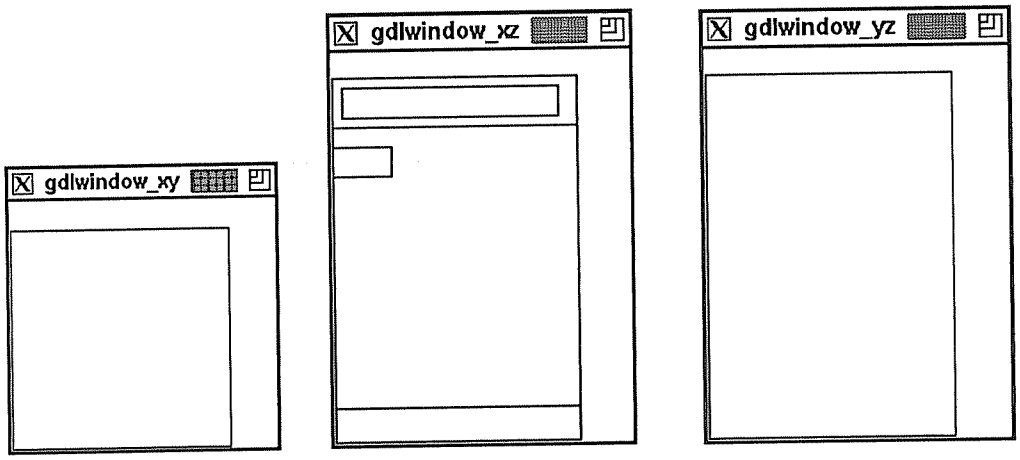
Below is a small example of a GDL program, which plots a dishwasher.

```
! Dishwasher   19.08.1989.
! internal macro: -

pars a = 2',b = 2',c =3 '
mulx a/2'
muly b/2'

prism 4,c,0,0,2',0,2',2',0,2'

rotx 90

lin_ 0,3 1/2",0,2',3 1/2",0

lin_ 0,c-5",0,2',c-5",0
poly 4,1",c-4",23",c-4",23",c-1",1",c-1"

poly 4,0,c-10",6",c-10",6",c-7",0,c-7"

del top
end
```

The program is parsed, and the parsed result is this list of statement:

```
io(pars([(var(a),0.62),(var(b),0.62),(var(c),0.93)]))
trans(mulx(operator(3,/,var(a),0.62)))
trans(muly(operator(3,/,var(b),0.62)))
three_dim(prism(4.0,var(c),
          [(0.0,0.0),(0.62,0.0),(0.62,0.62),(0.0,0.62)]))
trans(rotx(90.0))
two_dim(lin_(0.0,0.0875,0.0,0.62,0.0875,0.0))
two_dim(lin_(0.0,operator(4,-,var(c),0.125),0.0,0.62,
             operator(4,-,var(c),0.125),0.0))
two_dim(poly(4.0,[(0.025,operator(4,-,var(c),0.1)),
                  (0.575,operator(4,-,var(c),0.1)),
                  (0.575,operator(4,-,var(c),0.025)),
                  (0.025,operator(4,-,var(c),0.025))]))
two_dim(poly(4.0,[(0.0,operator(4,-,var(c),0.25)),
                  (0.15,operator(4,-,var(c),0.25)),
                  (0.15,operator(4,-,var(c),0.175)),
                  (0.0,operator(4,-,var(c),0.175))]))
trans(del(0))
flow(exit)
exit
```

And the result of interpreting the code above is these three views



References

[Ste86] I. Sterling, E. Shapiro, *The Art of Prolog, Advanced Programming Techniques*, the MIT Press, 1986

[Fol84] J.D. Foley, A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Company. 1984

Appendixes

A The GDL description

This appendix contains the description of the part of the GDL language that the parser can handle.

The following description is based on the 'GDL Reference Definition'. We have added some comments at some places, marked as '(Note: ....)', when we want to stress something about our implementation, and we only describe parts that we have implemented.

This description is valid for GDL files in ArchiCAD® version 3.xx.

1. GDL language

GDL (Geometric Description Language) is a three-dimensional, general-purpose geometric description language. The structure and syntax of a GDL program are similar to the BASIC language. A GDL program consists of statements which define geometric elements in the three-dimensional space, manipulate numeric data and character strings, handle variables and control the sequence of execution.

GDL contains 2D and 3D geometric basic elements which can serve as "bricks" to build up complex shapes.

The essential task of a GDL program is to describe a system of geometric objects, so the main "building bricks" of GDL are the geometric objects. There are two kinds of them: primitive and compound.

In order to minimize the number of parameters of geometric objects, each compound object is specified in a "local coordinate system" fixed to the object. For instance, a cylinder is defined by two data items (length and radius). It begins in the origin and is coaxial with the z-axis. In a description like this, the position of the objects can be defined by preceding coordinate transformations of the local coordinate system.

The geometric parameters of each body refer to the actual local coordinate system.

2. Structure of a GDL file

A GDL program consists of lines separated by line-separators (end_of_line characters). Each text line can contain a single statement. If the parameters exceed a line, you can continue writing them in a continuation line. A comma (`,`) in the last position indicates that the statement continues in the next line. After an exclamation mark (`!`) you can write any comment in the line.

Any line can start with a label. A label is an integer number followed by a colon (`:`). Labels are not checked for single occurrence. A jump to a label refers to the first occurrence of the given label from the beginning of the current file.

The GDL language is not case sensitive; uppercase and lowercase letters are not distinguished.

The end of a GDL program is denoted by an END or EXIT statement or the end of the file.

A GDL file may begin with a PARS directive.

It is possible to use a GDL object from another GDL file (calling macros).
(Note: this is not implemented in the parser.)

Blank lines can be inserted into a GDL program without any effect and any number of spaces can be used between the operands and operators. After statements and macro calls the use of a space is obligatory.

Excess parameters of a statement are ignored.

GDL macros may not exceed 400 lines and/or 32700 characters and each line must be shorter than 80 characters. (Note: These limitations are not checked in the parser.)

3. Syntactic elements

3.1 Letters, digits and special symbols

letters `a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|x|y|z|`
`A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|X|Y|Z`

digits `0|1|2|3|4|5|6|7|8|9|0`

special symbols <space>|_(underline)`|!|:|,|;|.|+|-|*|/|^|=|<|>|#|"|`
`(|)|`<end_of_line>

3.2 Variables

A GDL program can handle variables defined by their identifiers, numbers and character strings. Variables and constants have floating point values.

variable letter

(Note: It seems that a variable can also be a string of letters, when one takes a look in the GDL files submitted with the ArchiCAD® package, so we have incorporated this "feature".)

number {[+/-]integer_part[.]}
{[+/-][integer_part].fractional_part} [E[+/-]exponent]

where integer_part, fractional_part and exponent are sequence of digits (integer numbers).

Examples: `1.0, -5.5, 6, 9E25, .25, -.25`

(Note: The GDL files in ArchiCAD® may have their measures in feet and inches, denoted by ' and ". We have chosen to rewrite X'Y" to Z, where Z is in metres, and then look at Z as a number described above.)

string a sequence of any characters (except quotation mark, exclamation mark and end_of_line character) between quotation marks (").

Example: `"This is a string"`

3.3 Expressions

You can write compound expressions in GDL statements

expression
: An expression determines a numeric value. It can be composed of numbers, variables and function calls connected by operators. Round bracket pairs `()` (precedence 1) are used to override the default precedence of the operators.

  Examples: `z, 5.5, (+15), -x, a*(b+c), SIN(x+y)*z, a+r*COS(i*d)`

3.4 Operators

GDL works with the operators below in decreasing precedence. The evaluation of an expression begins with the highest precedence operators or (in the case of equality) from left to right.

3.4.1 Arithmetic operators

^ (or **)	Power of	precedence 2
*	Multiplication	precedence 3
/	Division	
MOD	Modulo (remaining part) `X mod Y = X - Y * int(X/Y)`	
+	Addition	precedence 4
-	Subtraction, unary minus	

3.4.2 Relational operators

=	Equal	precedence 5
<	Less than	
>	Greater than	
<=	Less than or equal	
>=	Greater than or equal	
<> (or #)	Not equal	

Relational operators compare the values of arithmetic expressions, and returns `1` for "true" and `0` for "false".

3.4.3 Boolean operators

AND	Logical and	precedence 6
OR	Logical or	precedence 7
EXOR	Logical exclusive or	precedence 8

Since GDL uses floating point numbers only, Boolean operators work with real numbers, so `0.0` means "false", while any other number means "true". The value of a logical expression is also real, i.e. `1.0` for "true" and `0.0` for "false".

3.5 Functions

GDL gives the possibility to use analytic and numeric mathematical functions.

3.5.1 Arithmetic functions

ABS(x)	Absolute value
INT(x)	Truncate function (e.g. `INT(-1.234) = -1`)

FRA(x)	Fractional part (e.g. FRA(-1.234) = -0.234)
SGN(x)	Sign function (-1 if x < 0, + 1 if x > 0 and 0 if x = 0)
SQR(x)	Square root of x

3.5.2 Circular functions

These functions use degrees for arguments (COS, SIN, TAN) and for return values (ACS, ASN, ATN).

ACS(x)	Arcus cosine (-1.0 <= x <= 1.0; 0 <= ACS(x) <= 180)
ASN(x)	Arcus sine (-1.0 <= x <= 1.0; -90 <= ASN(x) <= 90)
ATN(x)	Arcus tangent (-90 <= ATN(x) <= 90)
COS(x)	Cosine
SIN(x)	Sine
TAN(x)	Tangent

3.5.3 Transcendental functions

EXP(x)	e power x
LGT(x)	Base 10 logarithm of x
LOG(x)	Natural logarithm of x

3.5.4 Boolean functions

NOT(x)	Logical negation (0 if x <> 0 and 1 if x = 0)

3.5 Constant

PI	Ludolph's constant 3.1415926

4. GDL statements

4.1 Directives

The directives influence the interpretation of the following GDL statements. Their influence is in effect until the next directive or the end of the program.

- LET varnam = expr

Value assignment. The LET directive cannot be omitted. The variable will store the evaluated value of the expression.

- MODEL {WIRE}
  {SURFACE}
  {VOLUME}

The directive MODEL gives the representation mode in the current macro. The following models are possible:

WIRE: only wire frame model, no surfaces or volumes. Objects are transparent.

SURFACE: surface model with edges and surfaces, no volumes. objects can hide other objects.

VOLUME: volumetric model with edges, surfaces and volumes. Objects can hide other objects and after cutting them new surfaces may appear.

The default model is SURFACE

- PEN expr

Sets the actual color. 0 < expr <= 128, but colors are modulo 16 interpreted.

The default is PEN 1

- RADIUS rmin, rmax

Sets the smoothness for cylindrical elements. A circle of radius r is represented:
if r <= rmin then by a hexagon,
if r >= rmax then by a 36 edged polygon,
if rmin < r < rmax then by an
(6+30*(r-rmin)/(rmax-rmin)) -edged polygon.

- SFLINE n

Explicit contour line search. The next n lines are considered to be contour lines of a cylindrical element and treated together. The extreme lines of this element are calculated and drawn according to the actual projection. It must be used before the lines are defined.
(0 < n <= 36)
The CYLIND, CONE and ELLIPS statements imply the SFLINE directive.

4.2 Flow control statements

- FOR varnam=initial_value to end_value [step step_value]

This is the first statement in a for loop. If the step keyword and the step_value are missing it is assumed to be 1.0.

- NEXT varnam

This is the last statement in a for loop.

The loop variable varies from initial_value to end_value by the step_value increment (or decrement) in each execution of the body of the loop (statements between the FOR and NEXT statements). If the loop variable exceeds the value of end_value, the program executes the statement following the next statement.

The body of the loop runs at least once.

- IF expr1 {THEN} expr2
  {GOTO}
  {GOSUB}

This is the conditional jump statement. If the value of expr1 is 0 the command has no effect, otherwise the execution continues at label expr2.

examples:
```
IF i > j GOTO 200+i*j
IF a THEN 28
IF i > 0 GOSUB 9000
```

- GOTO expr

GOTO is the unconditional jump statement. The program executes a branch to the statement denoted by the value of the label expression.

- GOSUB expr

GOSUB is the internal subroutine call where expr is the entry point of the routine.

- `RETURN`

The execution returns from an internal subroutine.

- `{EXIT}`
  `{END}`

Marks the end of the current GDL macro. The program terminates or returns to the level above. It is possible to use several ends or exits in a GDL file.

4.3 Input-output statements

- `PARS prompt [=default_value][,prompt [=default_value]]...`

This is the symbolic parameter list declaration.

`prompt` is the symbolic name of the corresponding GDL variable (`a`, `b`, `c`, etc.) and `default_value` appears in the object parameters dialogue. The `default_value` is optional. If it is missing, `0` is assumed.

The `PARS` directive (if used) must be the first executable statement in the macro.

`PARS` has an effect only if the macro is processed as a "main" program, i.e. not called by another macro. In this case the 3D presentation program displays a dialogue with the prompts and default values to enter the parameters.
If the macro is called from another macro, the parameters following the macro name are transferred automatically and the PARS directive has no effect.

Example: `PARS Width = 1.2, Height = 3.1, No_of_arms = 5`

- `PRINT ["prompt",][expr][,expr]...`

`PRINT` prints out remarks and values of expressions. The prompt must be enclosed in double quotes. No continuation line is allowed.

Examples:
```
PRINT "loop-variable:", I
PRINT J, K
PRINT "Just a string"
```

4.4 Transformations

- `ORIGO x,y,z`

This primitive specifies a new absolute origin for the coordinate system relative to the previous absolute origin. All current transformations are discarded.
Equivalent to

```
DEL TOP
ADDX x
ADDY y
ADDZ z
```

Use this command only in main GDL segments. Do not use it in ArchiCAD® symbols!

- `{ADDX}`
  `{ADDY}`
  `{ADDZ} expr`

Move local coordinate system along the given axis by `expr`.

- `{ROTX}`
  `{ROTY}`
  `{ROTZ} expr`

Rotate local coordinate system around the given axis by `expr` degrees, counterclockwise.

- {MULX}
  {MULY}
  {MULZ} expr

Scale local coordinate system along the given axis. Negative expr means simultaneous mirroring.

- DEL {TOP}
  {expr}

Delete previous expr transformations. The local coordinate system moves back to a previous position. If less than n transformations were issued, only they are deleted. If expr = 0 all transformations of the active macro are deleted.
DEL TOP is equivalent to DEL 0.

4.6 Two dimensional elements

- LIN_ x1,y1,z1,x2,y2,z2

A line segment between the points p1 = x1, y1, z1 and p2 = x2, y2, z2.

- CIRCLE r

A circle in the x-y plane with its centre in the origin and with a radius of r.

- ARC r,alpha,beta

An arc in the x-y plane with its centre in the origin from angle alpha to beta with a radius of r.

- RECT a,b

A rectangle in the x-y plane with sides a and b.

- RECT_ a,b,e1,e2

A rectangle in the x-y plane with sides a and b. The horizontal edges are omitted and the vertical edges are shifted by e1 and e2, respectively.

- POLY n,x1,y1,...,xn,yn

A polygon with n edges in the x-y plane. The coordinates of node 1 are xi and yi.

- POLY_ n,x1,y1,mask1,...,xn,yn,maskn

Similar to the normal POLY statement, but any of the vertices can be omitted.
If maski = 0 the vertex starting from the (xi,yi) apex will be omitted. If maski = 1 the vertex will be shown.

4.7 Compound three dimensional elements

- BRICK a,b,c

The first corner of the block is in the local origin and the edges with lengths a, b and c are along the x-, y- and z-axis respectively.

- CYLIND l,r

Cylinder, coaxial with the z-axis with a height of 1 and a radius of r.

- CONE h,r1,r2,alpha1,alpha2

Frustum of a cone where alpha1 and alpha2 are angles of inclination of the end surfaces to the z axis, r1 and r2 are radiuses of the end-circles and h is the height along the z-axis.

A regular cone can be obtained with the following parameters:

```
CONE h,r,0,90,90
```

- `ELLIPS r,h`

Half ellipsoid. Its cross-section in the x-y plane is an origin centred circle with a radius of `r`. The length of the half axis along the z-axis is `h`.

A semi-sphere can be obtained with the following parameters:

```
ELLIPS r,r
```

- `PRISM n,h,x1,y1,...,xn,yn`

Regular prism with its base polygon in the x-y plane (see the parameters of `POLY`). The height along the z-axis is `h`.

- `PRISM_ n,h,x1,y1,mask1,...,xn,yn,maskn`

Similar to the `PRISM` statement, but any of the horizontal vertices and sides can be omitted.

The mask number is a four bit binary integer (between `0` and `15`). `j1 + j2 * 2 + j3 * 4 + j4 * 8`, where `j1`, `j2`, `j3` and `j4` can be `0` or `1`. The `j1`, `j2`, `j3` and `j4` numbers represent whether the vertices and the side are present (`1`) or omitted (`0`).

- `ELBOW r1,alpha,r2`

Segment elbow in the x-z plane. The radius of the arc is `r1`, the angle is `alpha` and radius of the tube segment is `r2`.

B The parser

```
%%% File:     gdlparser.pl
%%% Author:   Martin Aronsson

%%% We have used the DCG facility to parse a file.
%%% The parsing is divided into three passes, one pass reads the
%%% letters from the file,one pass for parsing
%%% letters into higher level symbols, and one pass for parsing
%%% the higher level symbols into statements.
%%% We have in general used the first argument of the DCG predicates
%%% to pass the answers.
%%% We have not optimized this code, but tried to make it readable.
%%% If we have succeeded in that I'm not sure!!
%%%----------------------


%%% read_to_eof reads all characters from the stream S until the end
%%% of file is reached. X contains the answer.
read_to_eof(S,X) :-
      get0(S,Z),
      (Z = -1 -> X = [eof] ; read_to_eof(S,R),X = [Z|R]).

%%%====================
%%% Test predicate which performs both parsing, pretty-printing
%%% and interpretation
rap(F) :- read_and_parse_file(F,S),!,pp(S),inter(S).

%%% Entry point for the parser. The file File is parsed, and the
%%% result is passed as a list in the second argument.
%%% The parser performs two passes after having read the file;
%%% first it parses the tokens, that is, symbols, numbers,
%%% parenthesis, operators etc. Then it parses the statements.
read_and_parse_file(File,Struct) :-
      open(File,read,Stream),
      undo(close(Stream)),
      write(' Compiling file '),write(File),nl,
      read_to_eof(Stream,ListOfLetters),
```

```
      parse_tokens(ListOfTokens,ListOfLetters,[]),!,
      write('End of pass 1'),nl,nl,
      parse_statements(Struct,ListOfTokens,[]),
      write('End of pass 2'),nl,nl.

%%%====================
%%% Parses tokens as character strings
parse_tokens([eof]) --> [eof],!.
parse_tokens(S) --> [32],!,parse_tokens(S).     % skip spaces
parse_tokens(S) --> [9],!,parse_tokens(S).      % skip tabs
parse_tokens(S) --> [33],!,parse_comment,       % skip comments
      parse_tokens(S).
parse_tokens([F|R]) --> token(F),!,parse_tokens(R).
parse_tokens(S) --> parse_error_to_eol,parse_tokens(S).

parse_error_to_eol --> parse_error_to_eol(X),
      {name(X1,X),
       write('Following line begins with a syntax error: '),
       nl,tab(2),write(X1),nl,
       write(' The line is skipped, which will probably create '),
       write('a statement error later on'),nl,nl}.

parse_error_to_eol([F|R]) --> [F],{\+(F = 10)},!,
      parse_error_to_eol(R).
parse_error_to_eol([]) --> [].

token(label(X)) --> label(X).
token(num(X)) --> number(X).
token(string(X)) --> string(X).
token(word(X)) --> word(X).
token(X) --> special_symbol(X).

parse_comment --> [10],!.                  % end of comment
parse_comment --> [_],parse_comment.

%%%--------------------
%%% It is a bit difficult to parse a number, since there are several
%%% possibilities. We do not care about negative
%%% numbers here as it is simpler to handle that later when parsing
%%% expressions. All other possibilities are handled here:
%%% - integers (for example 1, 34)
%%% - rational numbers (for example 1 2/3)
%%% - floating points (for example 2.5, 12.0e3)
%%% - since the unit here is meter, and the dimensions could be given
%%%   in feet and/or inches, we have to convert from feet and inches
%%%   to meters (for example 2' will be converted to 0.305 * 2 and 3"
%%%   will be converted to 0.0254 * 3). The construction 2'3" will be
%%%   treated when expressions are parsed in the third pass.

number(N1) --> num(N),{roundoff(N,6,N1)}.

num(N) --> sequence_of_digits(X),!,num11(N,X).      % The integer part
num(N) --> num1(N,0).                      % No integer part

num11(N,X) --> num1(N,X),!.                % rational or fractional part
num11(N,X) --> {name(X1,X)},unit(N,X1).    % Just integer part

num1(N,X) -->                              % Rational
      spaces,sequence_of_digits(T),!,"/",sequence_of_digits(B),
      {name(T1,T),name(B1,B),name(X1,X),N1 is X1 + (T1 / B1)},
      unit(N,N1).
num1(N,X) -->                              % fractional part
      ".",!,sequence_of_digits(F),         % with exponent
```

```
      exp(N1,X,F),unit(N,N1).

%%% exp parses the exponent, if there is one. It also converts the
%%% number collected sofar to a prolog number using name(X,Y)
exp(N,X,F) --> "e","-",!,sequence_of_digits(E),    % Negative exponent
      {append("e-",E,Tmp),append(F,Tmp,Tmp1),
       append(X,[46|Tmp1],N1),name(N,N1)}.
exp(N,X,F) --> "E","-",!,sequence_of_digits(E),    % Negative exponent
      {append("e-",E,Tmp),append(F,Tmp,Tmp1),
       append(X,[46|Tmp1],N1),name(N,N1)}.
exp(N,X,F) --> "e","+",!,sequence_of_digits(E),    % Positive exponent
      {append("e",E,Tmp),append(F,Tmp,Tmp1),
       append(X,[46|Tmp1],N1),name(N,N1)}.
exp(N,X,F) --> "E","+",!,sequence_of_digits(E),    % Positive exponent
      {append("e",E,Tmp),append(F,Tmp,Tmp1),
       append(X,[46|Tmp1],N1),name(N,N1)}.
exp(N,X,F) --> "e",!,sequence_of_digits(E),        % Positive exponent
      {append("e",E,Tmp),append(F,Tmp,Tmp1),
       append(X,[46|Tmp1],N1),name(N,N1)}.
exp(N,X,F) --> "E",!,sequence_of_digits(E),        % Positive exponent
      {append("e",E,Tmp),append(F,Tmp,Tmp1),
       append(X,[46|Tmp1],N1),name(N,N1)}.
exp(N,X,F) -->                                     % Just conversion from
      {append(X,[46|F],Tmp),name(N,Tmp)}.          % ascii-list to a number

sequence_of_digits([F|R]) --> digit(F),sequence_of_digits(R).
sequence_of_digits([X]) --> digit(X).

spaces --> " ",optional_spaces.
optional_spaces --> " ",!,optional_spaces.
optional_spaces --> [].

%%% parses a different and converts between foot/inch to meter.
unit(N,N1) --> [39],!,{N is N1 * 0.31}.        % ' (foot)
unit(N,N1) --> [34],!,{N is N1 * 0.025}. % " (inch)
unit(N,N) --> [].

%%%------------------
%%% Parses a string beginning and ending with a ".
string(X1) --> [34],string1(X),{name(X1,X)}.

string1([F|R]) --> [F],
      {\+ F = 39, \+ F = 34, \+ F = 33, \+ F = 10},
      string1(R).
string1([]) --> [34].
string1([]) --> {write('Syntax error: miss "'),nl,fail}.

%%%--------------------
%%% A word is a sequence of letters
word(X) --> letter(F),word1(R),{name(X,[F|R])},!.

word1([F|R]) --> letter(F),word1(R).
word1([95|R]) --> "_",word1(R).
word1([]) --> [].

%%%--------------------
%%% Labels are numbers followed by a ':'.
label(X1) --> sequence_of_digits(X),":",{name(X1,X)}.

%%%--------------------
%%% Definition of letters, digits and other characters,
%%% uppercase letters are converted to lowercase letters.
letter(X1) --> [X],
```

```
      {65 =< X, X =< 90,X1 is X + 32}.          % A < X < Z
letter(X) --> [X],{97 =< X, X =< 122}.          % a < X < z

digit(X) --> [X],{48 =< X, X =< 57}.            % 0 < X < 9

special_symbol(X1) --> [X],
      {member(X,"<>_=+-*/,;.#()^"),
       name(X1,[X])}.
special_symbol(lf) --> [10].                    % end_of_line
special_symbol(lf) --> [13].                    % end_of_line

%%% End of parsing tokens

%%%====================
%%% Parsing statements is the second pass of the overall parsing.
%%% Here we should organize symbols, numbers and such things into
%%% statements. We mark each statement with its "class name", i.e. a
%%% for-loop is a flow statement, so the resulting term is
%%% flow(for(....)).
parse_statements([exit]) --> [eof],!.
parse_statements(S) --> [lf],!,
      parse_statements(S).
parse_statements([label(X)|R]) --> [label(X)],!,
      parse_statements(R).
parse_statements([directive(F)|R]) --> directive(F),!,
      parse_statements(R).
parse_statements([flow(F)|R]) --> flow(F),!,
      parse_statements(R).
parse_statements([io(F)|R]) --> io(F),!,
      parse_statements(R).
parse_statements([trans(F)|R]) --> transformation(F),!,
      parse_statements(R).
parse_statements([two_dim(F)|R]) --> two_dim(F),!,
      parse_statements(R).
parse_statements([three_dim(F)|R]) --> three_dim(F),!,
      parse_statements(R).
parse_statements(S) --> parse_error,
      parse_statements(S).

%%% We want the statements in a loop to be a list of its own, and
%%% therefore we have to use a second predicate for this parsing.
%%% (In fact it is possible to use the same loop as for the main
%%%  program, the only difference is that the stop conditions are
%%%  different.)
parse_statements_in_loop(Sl,St,S) --> [lf],!,
      parse_statements_in_loop(Sl,St,S).
parse_statements_in_loop([label(X)|R]) --> [label(X)],!,
      parse_statements_in_loop(R).
parse_statements_in_loop(Sl,St,[directive(F)|R]) --> directive(F),!,
      parse_statements_in_loop(Sl,St,R).
parse_statements_in_loop(Sl,St,[flow(F)|R]) --> flow(F),!,
      parse_statements_in_loop(Sl,St,R).
parse_statements_in_loop(Sl,St,[io(F)|R]) --> io(F),!,
      parse_statements_in_loop(Sl,St,R).
parse_statements_in_loop(Sl,St,[trans(F)|R]) --> transformation(F),!,
      parse_statements_in_loop(Sl,St,R).
parse_statements_in_loop(Sl,St,[two_dim(F)|R]) --> two_dim(F),!,
      parse_statements_in_loop(Sl,St,R).
parse_statements_in_loop(Sl,St,[three_dim(F)|R]) --> three_dim(F),!,
      parse_statements_in_loop(Sl,St,R).
parse_statements_in_loop(SlVal,StVal,[flow(next(V,StVal,SlVal))]) -->
      [word(next),word(V),lf],!.
parse_statements_in_loop(SlVal,StepVal,S) -->
```

```
      parse_error,parse_statement_in_loop(SlVal,StepVal,S).

%%% When an error occurs, we print the erroneous statement
parse_error --> parse_to_eol(X),!,
      {write('Error in parsing statements: '),nl,tab(2),
       write_list(X),nl,tab(1),write(' is not a proper statement'),nl,
       write(' The statement is skipped'),nl,nl}.

%%% We have to remove our "typing" of the tokens to write them out
parse_to_eol([]) --> [lf],!.
parse_to_eol([F|R]) --> [X],{strip_functor(X,F)},parse_to_eol(R).

%%% This is used to strip of our structures, to print out
%%% the "clean" statement
strip_functor(X,X) :-                          % Use the same database
      name(X,X1),special_symbol(X,X1,[]).      % as before
strip_functor(X,F) :- arg(1,X,F).

%%% A directive is a statement that changes the global state
directive(let(Varnam,E)) -->
      [word(let),word(Varnam),'='],
      expression(E),[lf].
directive(model(M)) --> [word(model)],!,
      model(M),[lf].
directive(pen(E)) --> [word(pen)],
      expression(E),[lf].
directive(box(Xmin,Xmax,Ymin,Ymax,Zmin,Zmax)) --> [word(box)],
      expression(Xmin),[','],optional_lf,
      expression(Xmax),[','],optional_lf,
      expression(Ymin),[','],optional_lf,
      expression(Ymax),[','],optional_lf,
      expression(Zmin),[','],optional_lf,
      expression(Zmax),[lf].
directive(box(cont)) --> [word(box),word(cont)],[lf].
directive(box(pause)) --> [word(box),word(pause)],[lf].
directive(radius(Rmin,Rmax)) --> [word(radius)],
      expression(Rmin),[','],optional_lf,expression(Rmax),[lf].
directive(sfline(N)) --> [word(sfline)],expression(N),[lf].

model(wire) --> [word(wire)].
model(surface) --> [word(surface)].
model(volume) --> [word(volume)].
model(solid) --> [word(solid)].

%%% A flow statement directs the execution (loops etc).
flow(for(let(Varnam,StVal),SlVal,1.0,List)) --> [word(for)],
      [word(Varnam),'='],expression(StVal),
      [word(to)], expression(SlVal),[lf],
      parse_statements_in_loop(SlVal,1.0,List).
flow(for(let(Varnam,StVal),SlVal,StepVal,List)) --> [word(for)],
      [word(Varnam),'='],expression(StVal),
      [word(to)], expression(SlVal),
      [word(step)], expression(StepVal), [lf],
      parse_statements_in_loop(SlVal,StepVal,List).
flow(if_goto(Expr1,Expr2)) --> [word(if)],
      expression(Expr1),[word(then)],
      expression(Expr2),[lf].
flow(if_goto(Expr1,Expr2)) --> [word(if)],
      expression(Expr1),[word(goto)],
      expression(Expr2),[lf].
flow(if_gosub(Expr1,Expr2)) --> [word(if)],
      expression(Expr1),[word(gosub)],
      expression(Expr2),[lf].
```

```
flow(goto(Expr)) --> [word(goto)],
      expression(Expr),[lf].
flow(gosub(Expr)) --> [word(gosub)],
      expression(Expr),[lf].
flow(return) --> [word(return),lf].
flow(exit) --> [word(exit)].
flow(exit) --> [word(end)].

%%% The I/O primitives are straightforward
io(pars([F|R])) --> [word(pars)],
      pars(F),                              % parse first var
      io_pars1(R).                          % parse rest of vars
io(print(Prompt,List_of_exp)) --> [word(print)],
      [string(Prompt)],
      parse_expressions(List_of_exp).
io(print('',[Exp|List_of_exp])) --> [word(print)],
      expression(Exp),
      parse_expressions(List_of_exp).

parse_expressions([]) --> [lf].
parse_expressions([F|R]) --> [','],optional_lf,expression(F),
      parse_expressions(R).

pars((var(Var),Default)) --> [word(Var),'='],
      expression(Default).
pars((var(Var),0)) --> [word(Var)].

io_pars1([]) --> [lf].
io_pars1([F|R]) --> [','],optional_lf,pars(F),io_pars1(R).

%%%------------------
%%% Parsing expressions is a tricky thing, we have to avoid
%%% looping in the left branch of a binary expression. The predicate
%%% simple_term takes care of the terms that are not binary, for
%%% example numbers and unary functions. parse_couples parses binary
%%% operators until a simple expression is reached, and produces a
%%% pair with the left term together with the binary operator.
%%% combine takes the pairs and produces the correct representation
%%% according to the precedence values.
%%% GDL has no unary operators.
expression(X) --> parse_couples(X1),     % First parse to end_of_expr
                                         % then combine parsed
      {combine(start,X,X1,[])}.          % couples

simple_term(Z) --> [num(X),num(Y)],      % rational number
      {Z is X + Y}.
simple_term(X) --> [num(X)].
simple_term(X) --> ['+',num(X)].
simple_term(Y) --> ['-',num(X)],{Y is -X}.
simple_term(X) --> ['('],!,expression(X),[')'].
simple_term(constant(X)) --> constant(X).
simple_term(var(X)) --> variable(X).
simple_term(function(Name,Arg)) -->
      function(Name),['('],expression(Arg),[')'].

parse_couples([c(Term,Oper,N)|R]) -->
      simple_term(Term),operator(N,Oper),!,
      parse_couples(R).
parse_couples([end(Term)]) --> simple_term(Term).

%%% combine uses combine1 for traversing the pairs. The first argument
%%% to combine1 is an accumulating term, RT is to be instantiated
%%% depending on the next operator's precedence value.
```

```
combine(start,Answ) --> [end(Answ)],!.              % Just a simple term
combine(start,Answ) --> [c(Term,Oper,N)],!,
      combine1(operator(N,Oper,Term,RT),RT,Answ).

combine1(Sofar,Term,Sofar) --> [end(Term)].
combine1(Sofar,RT,Answ) --> [c(Term1,Oper,N1)],
      {insert(c(Term1,Oper,N1),Sofar,NSofar,RT,NRT)},
       combine1(NSofar,NRT,Answ).

%%% insert traverses the expression tree built sofar, and inserts
%%% the new Term-Operator pair in the tree according to the precedence
%%% value of the operator.
insert(c(Term1,Oper,N1),Var,operator(N1,Oper,Term1,Var1),Var,Var1) :-
      var(Var),!.
insert(c(Term1,Oper1,N1),operator(N,Oper,T1,T2),
       operator(N1,Oper1,operator(N,Oper,T1,T2),Var),Term1,Var) :-
      N =< N1,!.
insert(c(Term1,Op1,N1),operator(N,Op,T1,T2),operator(N,Op,T1,T2p),
       RT,NRT) :-
      N > N1,!,
      insert(c(Term1,Op1,N1),T2,T2p,RT,NRT).

%%% Transformation statements transforms the 3d-axes and origo
transformation(origo(X,Y,Z)) --> [word(origo)],
      expression(X),[','],optional_lf,
      expression(Y),[','],optional_lf,
      expression(Z),[lf].
transformation(addx(E)) --> [word(addx)],
      expression(E),[lf].
transformation(addy(E)) --> [word(addy)],
      expression(E),[lf].
transformation(addz(E)) --> [word(addz)],
      expression(E),[lf].
transformation(rotx(E)) --> [word(rotx)],
      expression(E),[lf].
transformation(roty(E)) --> [word(roty)],
      expression(E),[lf].
transformation(rotz(E)) --> [word(rotz)],
      expression(E),[lf].
transformation(mulx(E)) --> [word(mulx)],
      expression(E),[lf].
transformation(muly(E)) --> [word(muly)],
      expression(E),[lf].
transformation(mulz(E)) --> [word(mulz)],
      expression(E),[lf].
transformation(del(0)) --> [word(del),word(top),lf].
transformation(del(E)) --> [word(del)],expression(E),[lf].

%%% two_dim statements plots two dimensional objects, i.e. objects
%%% that can be represented in a two-dimensional plane.
two_dim(lin_(X1,Y1,Z1,X2,Y2,Z2)) --> [word(lin_)],
      expression(X1),[','],optional_lf,
      expression(Y1),[','],optional_lf,
      expression(Z1),[','],optional_lf,
      expression(X2),[','],optional_lf,
      expression(Y2),[','],optional_lf,
      expression(Z2),[lf].
two_dim(circle(R)) --> [word(circle)],
      expression(R),[lf].
two_dim(arc(R,Alpha,Beta)) --> [word(arc)],
      expression(R),[','],optional_lf,
      expression(Alpha),[','],optional_lf,
      expression(Beta),[lf].
```

```
two_dim(rect(A,B)) --> [word(rect)],
      expression(A),[','],optional_lf,
      expression(B),[lf].
two_dim(rect_(A,B,E1,E2)) --> [word(rect_)],
      expression(A),[','],optional_lf,
      expression(B),[','],optional_lf,
      expression(E1),[','],optional_lf,
      expression(E2),[lf].
two_dim(poly(N,List_of_coord)) -->
      [word(poly)],expression(N),
      parse_list_of_coord2(List_of_coord).
two_dim(poly_(N,List_of_coord)) --> [word(poly_)],
      expression(N),
      parse_list_of_coord3(List_of_coord).

%%% Three dimensional statements plots objects that are compound
%%% of other objects
three_dim(brick(A,B,C)) --> [word(brick)],
      expression(A),[','],optional_lf,
      expression(B),[','],optional_lf,
      expression(C),[lf].
three_dim(cylind(L,R)) --> [word(cylind)],
      expression(L),[','],optional_lf,
      expression(R),[lf].
three_dim(cone(H,R1,R2,Alpha1,Alpha2)) --> [word(cone)],
      expression(H),[','],optional_lf,
      expression(R1),[','],optional_lf,
      expression(R2),[','],optional_lf,
      expression(Alpha1),[','],optional_lf,
      expression(Alpha2),[lf].
three_dim(ellips(H,R)) --> [word(ellips)],
      expression(H),[','],optional_lf,
      expression(R),[lf].
three_dim(prism(N,H,List_of_coord)) --> [word(prism)],
      expression(N),[','],optional_lf,
      expression(H),
      parse_list_of_coord2(List_of_coord).
three_dim(prism_(N,H,List_of_coord)) --> [word(prism_)],
      expression(N),[','],optional_lf,
      expression(H),
      parse_list_of_coord3(List_of_coord).
three_dim(elbow(R1,Alpha,R2)) --> [word(elbow)],
      expression(R1),[','],optional_lf,
      expression(Alpha),[','],optional_lf,
      expression(R2),[lf].

%%% This predicate parses a list of couples
parse_list_of_coord2([]) --> [lf].
parse_list_of_coord2([(X,Y)|R]) -->
      [','],optional_lf,expression(X),
      [','],optional_lf,expression(Y),
      parse_list_of_coord2(R).

%%% This predicate parses a list of three expressions
parse_list_of_coord3([]) --> [lf].
parse_list_of_coord3([(X,Y,Mask)|R]) -->
      [','],optional_lf,expression(X),
      [','],optional_lf,expression(Y),
      [','],optional_lf,expression(Mask),
      parse_list_of_coord3(R).

%%% This predicate parses a list of four expressions
parse_list_of_coord4([]) --> [lf].
```

```
parse_list_of_coord4([(X,Y,Z,Mask)|R]) -->
      [','],optional_lf,expression(X),
      [','],optional_lf,expression(Y),
      [','],optional_lf,expression(Z),
      [','],optional_lf,expression(Mask),
      parse_list_of_coord4(R).

%%% This predicate parses a list of four expressions, and also returns
%%% the length of the list
parse_list_of_n_coord([],0) --> [].
parse_list_of_n_coord([(X,Y,Z,W)|R],N) --> {N > 0,N1 is N - 1},
      [','],optional_lf,expression(X),
      [','],optional_lf,expression(Y),
      [','],optional_lf,expression(Z),
      [','],optional_lf,expression(W),
      parse_list_of_n_coord(R,N1).

%%%-------------------
%%% Discards linefeeds that are not statement deliminiters
optional_lf --> [lf],!,optional_lf.
optional_lf --> [].

%%%-------------------
%%% Some constants
constant(pi) --> [word(pi)].
constant(bas) --> [word(bas)].
%constant(nod) --> [word(nod)].                    % Are not used
%constant(lin) --> [word(lin)].                    % Are not used
%constant(tri) --> [word(tri)].                    % Are not used
%constant(tet) --> [word(tet)].                    % Are not used

%%%-------------------
%%% A variable is a word
variable(X) --> [word(X)].

%%%===================
%%% Each operator has a precedence value attached to it
operator(2,pow) --> ['^'].
operator(2,pow) --> ['*','*'],!.
operator(3,'*') --> ['*'].
operator(3,'/') --> ['/'].
operator(3,mod) --> [word(mod)].
operator(4,'+') --> ['+'].
operator(4,'-') --> ['-'].

operator(5,'<=') --> ['<','='],!.
operator(5,'>=') --> ['>','='],!.
operator(5,'<>') --> ['<','>'],!.
operator(5,'<>') --> ['#'].
operator(5,'=') --> ['='].
operator(5,'<') --> ['<'].
operator(5,'>') --> ['>'].

operator(6,and) --> [word(and)].
operator(7,or) --> [word(or)].
operator(8,xor) --> [word(exor)].

%%%-------------------
%%% Listing the possible functions

function(not) --> [word(not)].

function(abs) --> [word(abs)].
```

```
function(int) --> [word(int)].
function(fraction) --> [word(fra)].
function(sgn) --> [word(sgn)].
function(sqrt) --> [word(sqr)].

function(acos) --> [word(acs)].
function(asin) --> [word(asn)].
function(atan) --> [word(atn)].
function(cos) --> [word(cos)].
function(sin) --> [word(sin)].
function(tan) --> [word(tan)].

function(exp) --> [word(exp)].
function(lgt) --> [word(lgt)].
function(log) --> [word(log)].

%%%=====================
%%% Some auxiliary predicates

append([],P,P).
append([F|R],L,[F|R1]) :- append(R,L,R1).

member(E,[E|_]).
member(E,[_|R]) :- member(E,R).

roundoff(X,Dec,Answ) :- Answ is round(X * pow(10,Dec)) / pow(10,Dec).

write_list([]).
write_list([F|R]) :- write(F),tab(1),write_list(R).

%%% Pretty-print routine
pp([]) :- nl.
pp([F|R]) :- write(F),nl,pp(R).
```

C The interpreter

```
%%% File:     inter1.pl
%%% Author:   Martin Aronsson

%%% This file contains an interpreter for part of the structure,
%%% that is the output from the gdl parser. The interpreter implements
%%% not all possible commands.
%%% The program is represented by a list, containing the statements.
%%% The interpreter opens three windows, where the planes x-y, x-z and
%%% y-z are plotted. When the interpreter reaches a "plotting"
%%% statement, it plots the lines in the three views according to the
%%% statement. When the "EXIT" statement is reached, the interpreter
%%% stops and waits for a carrige return. It then removes the windows
%%% and the execution terminates.
%%% This interpreter is not intended as a complete implementation of
%%% GDL, but an example of how an interpreter can be written, and also
%%% for running some GDL example programs.
%%%
%%% We would like to emphasize that this program can be much more
%%% efficiently implemented. This is not an example of a very
%%% efficient interpreter, but an understandable program (at least we
%%% hope so). The interpreter uses the ISP math library and the ISP
%%% graphics manager.

%%% We just want to load the modules once, therefore we must know when
%%% we have already loaded them (i.e. if the file is reloaded later)
:- (current_predicate(modules_are_loaded,_) -> true ;
```

```
      use_module(library(math)),use_module(library(gmlib)),
      start, assert(modules_are_loaded)).
:- dynamic my_window/3.

%%% General routine for open a view in a window, and store the name
%%% in an internal database.
open_window(Name) :- my_window(Name,_,_),!.
open_window(Name) :-
      gmcreate(View,view(500,250)),
      gmcreate(W,window(Name,View)),
      findall(foo,my_window(_,_,_),L),
      length(L,Num),
      Y is Num * 300,
      gmsend(W,open(500,Y)),!,
      assert(my_window(Name,View,W)).

%%% Close a window
close_window(Name) :-
      retract(my_window(Name,_,W)),
      gmsend(W,close),!.

%%% Close all user-created windows
close_all_windows :-
      retract(my_window(_,_,W)),
      W => close,fail.
close_all_windows.

%%%-----------------
%%% The entry point to the interpreter. First the three views are
%%% opened, the undo predicate ensures that the windows always will
%%% be closed, and then we call the actual interpreter.
inter(List) :-
      open_window(gdlwindow_xy),
      open_window(gdlwindow_xz),
      open_window(gdlwindow_yz),
      undo(close_all_windows),
      inter(List,List,[],
            [[[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]]],[]).

%%% The interpreter. The first element in the first argument's
%%% list contains the statement to be executed. The second argument
%%% contains the whole program, and is used when a jump should take
%%% place. The third argument contains the variable environment, which
%%% is an association list (i.e. a list of variable-value pairs). The
%%% fourth argument contains the current graphical state, i.e. the
%%% axis' lengths, orientation, scales etc. It is represented by a 4
%%% row 4 column matrix in the usual way (see a graphical handbook,
%%% for example Foley & van Dam, Fundamentals of Interactive Computer
%%% Graphics, for a detailed explanation of how to use matrixes for
%%% graphical representation).
%%% The matrixes are represented as lists, [row1, row2, ..., rown].
%%% The fifth argument is a stack, where adresses and other things are
%%% pushed, for example when a GOSUB instruction is executed.
inter([flow(exit)|_],_,Env,_,_) :-
      output('Execution finished.',[],Env),!.
inter([label(_)|R],Prog,Env,State,Stack) :-!,    % Skip labels
      inter(R,Prog,Env,State,Stack).
inter([flow(X)|R],Pr,E,St,Stack) :- !,  % A flow stmnt jumps somewhere
      flow_int(X,R,E,Pr,St,Stack).      % so our recursion stops
inter([io(F)|R],Prog,Env,State,Stack) :- !,
      io_int(F,Env,Newenv),
      inter(R,Prog,Newenv,State,Stack).
inter([trans(F)|R],Prog,Env,State,Stack) :- !,
```

```
      trans_int(F,Env,State,Newstate),
      inter(R,Prog,Env,Newstate,Stack).
inter([directive(F)|R],Prog,Env,State,Stack) :- !,
      directive_int(F,Env,Env1),
      inter(R,Prog,Env1,State,Stack).
inter([two_dim(F)|R],Prog,Env,State,Stack) :- !,
      two_dim_int(F,Env,State),
      inter(R,Prog,Env,State,Stack).
inter([three_dim(F)|R],Prog,Env,State,Stack) :- !,
      three_dim_int(F,Env,State),
      inter(R,Prog,Env,State,Stack).
inter([F|R],Prog,Env,State,Stack) :-
      write('Internal error, the internal statement '),nl,
      tab(2),write(F),nl,
      write(' is not recognized by the interpreter, and skipped'),
      nl,nl,
      inter(R,Prog,Env,State,Stack).

%%% We have only implemented the directiv LET. All other directives
%%% are skipped.
directive_int(let(V,Expr),Env,Env1) :- !,
      evaluate(Expr,E,Env),
      make_env(V,E,Env,Env1).
directive_int(_,Env,Env).

%%% A flow statement jumps somewhere in the code. We here need the
%%% whole program (the second argument of inter), to find the label
%%% where the execution is to be resumed.
flow_int(if_goto(Expr1,Expr2),R,Env,Prog,State,Stack) :- !,
      evaluate(Expr1,Expr11,Env),
      (1 is integer(Expr11) ->                  % if-part true?
       evaluate(Expr2,Expr21,Env),
       Expr is integer(Expr21),
       append(_,[label(Expr)|Cont],Prog),       % find label
       inter(Cont,Prog,Env,State,Stack) ;       % resume execution
       inter(R,Prog,Env,State,Stack)).          % otherwise go on

flow_int(next(V,StepVal,SlVal),[],Env,Prog,
         State,[[_,R]|Stack]) :-
      evaluate(operator(4,+,var(V),StepVal),
               Val,Env),                        %next value in for-loop
      make_env(V,Val,Env,NewEnv),
      SlVal < Val,!,                            % end of loop?
      inter(R,Prog,NewEnv,State,Stack).         % pop R from
                                                % stack,resume exec.
flow_int(next(V,StepVal,_),[],Env,Prog,State,
         [[L,R]|Stack]) :- !,
      evaluate(operator(4,+,var(V),StepVal),
               Val,Env),                        % not end of loop
      make_env(V,Val,Env,NewEnv),
      inter(L,Prog,NewEnv,State,[[L,R]|Stack]). % resume at loop's
                                                 % start
flow_int(gosub(Expr),R,Env,Prog,State,Stack) :- !,
      evaluate(Expr,L,Env),
      Label is integer(L),
      append(_,[label(Label)|Cont],Prog),       % find label
      inter(Cont,Prog,Env,State,[R|Stack]).     % push current PC
                                                % and jump
flow_int(goto(Expr),_,Env,Prog,State,Stack) :- !,
      evaluate(Expr,L,Env),
      Label is integer(L),
      append(_,[label(Label)|Cont],Prog),       % find label
      inter(Cont,Prog,Env,State,Stack).         % jump!
```

```
flow_int(return,_,Env,Prog,State,[Cont|Stack]) :- !, % pop stack
      inter(Cont,Prog,Env,State,Stack).        % and resume execution
flow_int(for(let(Varnam,StVal),SlVal,N,List),
         Cont,Env,Prog,State,Stack) :- !,
      evaluate(StVal,StVal1,Env),              % For-loop, execute List
      make_env(Varnam,StVal1,Env,Env1),        % push List and Cont on
                                               % stack
      inter(List,Prog,Env1,State,
            [[List,Cont]|Stack]).              % resume execution

%%% Just two I/O primitives, PARS changes the environment, PRINT just
%%% writes things out.
io_int(pars(List),Env,NewEnv) :- !,
      input(List,Vals),
      bind_vars(List,Vals,Env,NewEnv).
io_int(print(Prompt,List_of_expr),Env,Env) :-
      output(Prompt,List_of_expr,Env).

bind_vars([],[],E,E).
bind_vars([(var(V),_)|R],[F|R1],Env,NEnv) :-
      make_env(V,F,Env,TmpEnv),
      bind_vars(R,R1,TmpEnv,NEnv).

%%% Create a fancy parse directive, with buttons etc.
input(List,Vals) :-
      construct_hboxes(List,Hboxes,Lin),
      gmcreate(Box1,vbox(Hboxes)),
      B1 <= button('OK',ok),
      B2 <= button('Abort',abort),
      gmcreate(O,output('_____________________________________')),
      gmcreate(Box2,hbox([B1,space,B2])),
      gmcreate(Box3,vbox([Box1,frame(space(Box2)),O])),
      gmcreate(W,window('input',Box3)),
      gmsend(W,open(400,400)),
      repeat,
      waitevent(E),                        % wait for button-down-event
      event_handle(E,W),                   % handle event
      get_input_vals(Lin,List,Vals,O),     % form a list of vals
      gmsend(W,close),!.

%%% Constructs the lines containing the leading text and the input
text field
construct_hboxes([],[],[]).
construct_hboxes([(var(V),D)|R],[F|R1],[In|R2]) :-
      name(V,V1),
      append("Give value for variable: ",V1,Text),
      gmcreate(Out,output(Text)),
      gmcreate(In,input(D)),
      gmcreate(F,hbox([Out,frame(In)])),
      construct_hboxes(R,R1,R2).

%%% A very simple event handle routine, just two buttons
event_handle(button(_,ok),_) :- !.
event_handle(button(_,abort),W) :- !,gmsend(W,close),abort.

%%% get the values from the different input fields, check them
get_input_vals([],_,[],_).
get_input_vals([F|R],[(var(V),_)|R1],[F2|R2],O) :-
      gmsend(F,in(T)),
      name(F2,T),
      (number(F2) -> get_input_vals(R,R1,R2,O) ;
       name(V,V1),append("Value must be a number: ",V1,Text),
       gmsend(O,out(Text)),fail).
```

```
%%% Creating a fancy output primitive
output(Prompt,List,Env) :-
      create_output(List,String,Env),
      name(Prompt,Prompt1),
      append(Prompt1,[32|String],MessageList),
      name(Message,MessageList),
      B1 <= button('OK',ok),
      B2 <= button('Abort',abort),
      gmcreate(O,output(Message)),
      gmcreate(Box2,hbox([B1,space,B2])),
      gmcreate(Box3,vbox([O,frame(space(Box2))])),
      gmcreate(W,window('output',Box3)),
      gmsend(W,open(400,400)),
      repeat,
      waitevent(E),
      event_handle(E,W),
      gmsend(W,close),!.

create_output([],[],_).
create_output([F|R],Answ,Env) :-
      evaluate(F,F2,Env),
      name(F2,F1),
      create_output(R,R1,Env),
      append(F1,[32|R1],Answ).

%%% The transformation statements, which transforms one state into
%%% another. We have used the following notation for matrix elements:
%%% A21 is the old element in the second row and first column,
%%% A21p is the new element. The state argument is a stack, so
%%% when a new state is created, it is pushed on top of the state
%%% stack and when a DEL statement is reached, the right number of
%%% states can be pushed off.
trans_int(origo(X,Y,Z),_,[FS|States],[NewState,FS|States]) :- !,
      NewState = [[1,0,0,0],[0,1,0,0],
                  [0,0,1,0],[X,Y,Z,1]].
trans_int(addx(Expression),Env,[FS|States],[NewState,FS|States]) :- !,
      evaluate(Expression,E,Env),
      FS = [[A11,A12,A13,A14],[A21,A22,A23,A24],
            [A31,A32,A33,A34],[A41,A42,A43,A44]],
      A41p is A41 + E * A11,
      NewState = [[A11,A12,A13,A14],[A21,A22,A23,A24],
                  [A31,A32,A33,A34],[A41p,A42,A43,A44]].
trans_int(addy(Expression),Env,[FS|States],[NewState,FS|States]) :- !,
      evaluate(Expression,E,Env),
      FS = [[A11,A12,A13,A14],[A21,A22,A23,A24],
            [A31,A32,A33,A34],[A41,A42,A43,A44]],
      A42p is A42 + E * A22,
      NewState = [[A11,A12,A13,A14],[A21,A22,A23,A24],
                  [A31,A32,A33,A34],[A41,A42p,A43,A44]].
trans_int(addz(Expression),Env,[FS|States],[NewState,FS|States]) :- !,
      evaluate(Expression,E,Env),
      FS = [[A11,A12,A13,A14],[A21,A22,A23,A24],
            [A31,A32,A33,A34],[A41,A42,A43,A44]],
      A43p is A43 + E * A33,
      NewState = [[A11,A12,A13,A14],[A21,A22,A23,A24],
                  [A31,A32,A33,A34],[A41,A42,A43p,A44]].
trans_int(mulx(Expression),Env,[FS|States],[NewState,FS|States]) :- !,
      evaluate(Expression,Xscale,Env),
      matrixmult([[Xscale,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]],
                 FS,NewState).
trans_int(muly(Expression),Env,[FS|States],[NewState,FS|States]) :- !,
```

```
      evaluate(Expression,Yscale,Env),
      matrixmult([[1,0,0,0],[0,Yscale,0,0],[0,0,1,0],[0,0,0,1]],
            FS,NewState).
trans_int(mulz(Expression),Env,[FS|States],[NewState,FS|States]) :- !,
      evaluate(Expression,Zscale,Env),
      matrixmult([[1,0,0,0],[0,1,0,0],[0,0,Zscale,0],[0,0,0,1]],
            FS,NewState).
trans_int(rotx(Expression),Env,[FS|States],[NewState,FS|States]) :- !,
      evaluate(Expression,Degrees,Env),
      Rad is (3.1415926 / 180) * Degrees,
      matrixmult([[1,0,0,0],[0,cos(Rad),sin(Rad),0],
            [0,-sin(Rad),cos(Rad),0],[0,0,0,1]],FS,NewState).
trans_int(roty(Expression),Env,[FS|States],[NewState,FS|States]) :- !,
      evaluate(Expression,Degrees,Env),
      Rad is (3.1415926 / 180) * Degrees,
      matrixmult([[cos(Rad),0,-sin(Rad),0],[0,1,0,0],
            [sin(Rad),0,cos(Rad),0],[0,0,0,1]],FS,NewState).
trans_int(rotz(Expression),Env,[FS|States],[NewState,FS|States]) :- !,
      evaluate(Expression,Degrees,Env),
      Rad is (3.1415926 / 180) * Degrees,
      matrixmult([[cos(Rad),sin(Rad),0,0],
                                                              [ -
sin(Rad),cos(Rad),0,0],[0,0,1,0],[0,0,0,1]],FS,NewState).
trans_int(del(0),Env,State,[Origstate]) :- !,
      append(_,[Origstate],State).
trans_int(del(N),Env,State,State1) :- !,
      evaluate(N,N1,Env),N2 is integer(N1),
      length(L,N2),
      append(L,State1,State).

%%% The plotting statements of the category two dimensional elements.
%%% The evaluate_coord-calls evaluates the points in the local
%%% coordinate system, and the plot-procedures transforms the local
%%% points into the global coordinate system.
two_dim_int(poly(_,List_of_coord),Env,State) :- !,
      evaluate_coord2_list(List_of_coord,Xvals,Yvals,Env),
      plot_poly(0,Xvals,Yvals,State).
two_dim_int(poly_(_,List_of_coord),Env,State) :- !,
      evaluate_coord2_list_mask(List_of_coord,Xvals,Yvals,Masks,Env),
      plot_poly_with_mask(0,Masks,Xvals,Yvals,State,Env).
two_dim_int(lin_(X1,Y1,Z1,X2,Y2,Z2),Env,S) :- !,
      evaluate(X1,Xv1,Env),                  % Evaluate local coordinates
      evaluate(Y1,Yv1,Env),
      evaluate(Z1,Zv1,Env),
      evaluate(X2,Xv2,Env),
      evaluate(Y2,Yv2,Env),
      evaluate(Z2,Zv2,Env),
      get_coord(Xv1,Yv1,Zv1,S,Xval1,Yval1,Zval1), % Get global coord.
      get_coord(Xv2,Yv2,Zv2,S,Xval2,Yval2,Zval2),
      my_window(gdlwindow_xy,V1,_),              % Plot the lines
      gmsend(V1,line(Xval1,Yval1,Xval2,Yval2)),  % in the views
      my_window(gdlwindow_xz,V2,_),
      gmsend(V2,line(Xval1,Zval1,Xval2,Zval2)),
      my_window(gdlwindow_yz,V3,_),
      gmsend(V3,line(Yval1,Zval1,Yval2,Zval2)).
two_dim_int(circle(R),Env,S) :- !,             % Done by a polygon
      evaluate(R,RVal,Env),
      get_n_coord(0,RVal,S,Xvals,Yvals,Zvals), % get the global coord.
      my_window(gdlwindow_xy,V1,_),            % plot the polygon
      gmsend(V1,polygon(Xvals,Yvals)),
      my_window(gdlwindow_xz,V2,_),
      gmsend(V2,polygon(Xvals,Zvals)),
      my_window(gdlwindow_yz,V3,_),
```

```
      gmsend(V3,polygon(Yvals,Zvals)).
two_dim_int(rect_(A,B,E1,E2),Env,S) :- !,
      evaluate(A,A1,Env),                          % evaluate local coord
      evaluate(B,B1,Env),
      evaluate(E1,E11,Env),
      evaluate(E2,E21,Env),
      Tmp is A1 - E21,
      get_coord(E11,0,0,S,X1,Y1,Z1),               % get global coord.
      get_coord(E11,B1,0,S,X2,Y2,Z2),
      get_coord(Tmp,0,0,S,X3,Y3,Z3),
      get_coord(Tmp,B1,0,S,X4,Y4,Z4),
      my_window(gdlwindow_xy,V1,_),                % plot lines
      gmsend(V1,line(X1,Y1,X2,Y2)),
      gmsend(V1,line(X3,Y3,X4,Y4)),
      my_window(gdlwindow_xz,V2,_),
      gmsend(V2,line(X1,Z1,X2,Z2)),
      gmsend(V2,line(X3,Z3,X4,Z4)),
      my_window(gdlwindow_yz,V3,_),
      gmsend(V3,line(Y1,Z1,Y2,Z2)),
      gmsend(V3,line(Y3,Z3,Y4,Z4)).
two_dim_int(rect(A,B),Env,S) :- !,
      evaluate(A,A1,Env),                          % evaluate local coord
      evaluate(B,B1,Env),
      get_coord(0,0,0,S,X1,Y1,Z1),                 % get global coord.
      get_coord(A1,0,0,S,X2,Y2,Z2),
      get_coord(0,B1,0,S,X3,Y3,Z3),
      get_coord(A1,B1,0,S,X4,Y4,Z4),
      my_window(gdlwindow_xy,V1,_),                % plot lines
      gmsend(V1,line(X1,Y1,X2,Y2)),
      gmsend(V1,line(X1,Y1,X3,Y3)),
      gmsend(V1,line(X3,Y3,X4,Y4)),
      gmsend(V1,line(X2,Y2,X4,Y4)),
      my_window(gdlwindow_xz,V2,_),
      gmsend(V2,line(X1,Z1,X2,Z2)),
      gmsend(V2,line(X1,Z1,X3,Z3)),
      gmsend(V2,line(X3,Z3,X4,Z4)),
      gmsend(V2,line(X2,Z2,X4,Z4)),
      my_window(gdlwindow_yz,V3,_),
      gmsend(V3,line(Y1,Z1,Y2,Z2)),
      gmsend(V3,line(Y1,Z1,Y3,Z3)),
      gmsend(V3,line(Y3,Z3,Y4,Z4)),
      gmsend(V3,line(Y2,Z2,Y4,Z4)).
two_dim_int(arc(R,Alpha,Beta),Env,S) :- !,
      evaluate(R,Rval,Env),
      evaluate(Alpha,Alpha1,Env),
      evaluate(Beta,Beta1,Env),
      get_n_coord_for_arc(0,Rval,Alpha1,Beta1,S,Xvals,Yvals,Zvals),
      my_window(gdlwindow_xy,V1,_),                % Plot arc
      draw_arc(V1,Xvals,Yvals),
      my_window(gdlwindow_xz,V2,_),
      draw_arc(V2,Xvals,Zvals),
      my_window(gdlwindow_yz,V3,_),
      draw_arc(V3,Yvals,Zvals).
two_dim_int(X,_,_) :-
      write('Internal error, the internal two '),
      write('dimensional statement'),nl,
      tab(2),write(X),nl,
      write(' is not recognized by the interpreter, '),
      write('dimensional statement and skipped'),nl,nl.

%%% Help procedure, which plots an arc around the origo
draw_arc(_,[_],[_]) :- !.
draw_arc(View,[X1,X2|Xs],[Y1,Y2|Ys]) :-
```

```
      gmsend(View,line(X1,Y1,X2,Y2)),
      draw_arc(View,[X2|Xs],[Y2|Ys]).

%%% Approximates the arc with 20 lines
get_n_coord_for_arc(H,Rval,Alpha,Beta,S,Xvals,Yvals,Zvals) :-
      get_n_coord1(H,20,20,Rval,S,Xvals,Yvals,Zvals,Alpha,Beta),!.

%%% Approximates the circle with 36 lines
get_n_coord(H,Rval,S,Xvals,Yvals,Zvals) :-
      get_n_coord1(H,36,36,Rval,S,Xvals,Yvals,Zvals,0,360),!.

%%% Evaluates a local point on the circle/arc, then transform it to
%%% global coordinates, and recurses
get_n_coord1(H,0,_,_,_,[],[],[],Startangle,Endangle).
get_n_coord1(H,N,M,Rval,S,[Rx|R1],[Ry|R2],[Rz|R3],Alpha,Beta) :-
      Rx1 is Rval * cos(((((Beta - Alpha) / M) * N) + Alpha) *
                  (6.2831852 / 360)),
      Ry1 is Rval * sin(((((Beta - Alpha) / M) * N) + Alpha) *
                  (6.2831852 / 360)),
      get_coord(Rx1,Ry1,H,S,Rx,Ry,Rz),
      N1 is N - 1,
      get_n_coord1(H,N1,M,Rval,S,R1,R2,R3,Alpha,Beta).

%%% Compound three dimensional objects have lots of coordinates, and
%%% therefore the procedures get large, but the principles are the
%%% same as for two-dimensional coordinates.
%%% We have just implemented three instructions.
three_dim_int(prism(_,H,List),Env,State) :- !,
      evaluate(H,H1,Env),
      evaluate_coord2_list(List,Xvals,Yvals,Env),
      plot_poly(0,Xvals,Yvals,State),
      plot_poly(H1,Xvals,Yvals,State),
      plot_prism_lines(List,H,Env,State).
three_dim_int(prism_(_,H,List),Env,State) :- !,
      evaluate(H,H1,Env),
      evaluate_coord2_list_mask(List,Xvals,Yvals,Masks,Env),
      decompress_mask(Masks,Masks1,Masks2,Masks3),
      plot_poly_with_mask(0,Masks1,Xvals,Yvals,State,Env),
      plot_poly_with_mask(H1,Masks3,Xvals,Yvals,State,Env),
      plot_prism_lines_masked(List,Masks2,H,Env,State).
three_dim_int(cylind(L,R),Env,S) :- !,
      evaluate(L,L1,Env),evaluate(R,R1,Env),
      get_n_coord(0,R1,S,Xvals,Yvals,Zvals),
      get_n_coord(L1,R1,S,Xvals1,Yvals1,Zvals1),
      get_coord(0,0,0,S,X1,Y1,Z1),
      length(Xvals,Len),

      find_extremum_points(X1,Y1,0,0,Xvals,Yvals,1,N),
      pick_the_extremum_points(N,Len,Xvals,Yvals,Xext1,Yext1,
                               Xext2,Yext2),
      pick_the_extremum_points(N,Len,Xvals1,Yvals1,Xex1,Yex1,
                               Xex2,Yex2),
      my_window(gdlwindow_xy,V1,_),
      gmsend(V1,polygon(Xvals,Yvals)),
      gmsend(V1,polygon(Xvals1,Yvals1)),
      gmsend(V1,line(Xext1,Yext1,Xex1,Yex1)),
      gmsend(V1,line(Xext2,Yext2,Xex2,Yex2)),

      find_extremum_points(X1,Z1,0,0,Xvals,Zvals,1,N1),
      pick_the_extremum_points(N1,Len,Xvals,Zvals,Xext3,Zext3,
                               Xext4,Zext4),
      pick_the_extremum_points(N1,Len,Xvals1,Zvals1,Xex3,Zex3,
                               Xex4,Zex4),
```

```
        my_window(gdlwindow_xz,V2,_),
        gmsend(V2,polygon(Xvals,Zvals)),
        gmsend(V2,polygon(Xvals1,Zvals1)),
        gmsend(V2,line(Xext3,Zext3,Xex3,Zex3)),
        gmsend(V2,line(Xext4,Zext4,Xex4,Zex4)),

        find_extremum_points(Y1,Z1,0,0,Yvals,Zvals,1,N2),
        pick_the_extremum_points(N2,Len,Yvals,Zvals,Yext5,Zext5,
                                 Yext6,Zext6),
        pick_the_extremum_points(N2,Len,Yvals1,Zvals1,Yex5,Zex5,
                                 Yex6,Zex6),
        my_window(gdlwindow_yz,V3,_),
        gmsend(V3,polygon(Yvals,Zvals)),
        gmsend(V3,polygon(Yvals1,Zvals1)),
        gmsend(V3,line(Yext5,Zext5,Yex5,Zex5)),
        gmsend(V3,line(Yext6,Zext6,Yex6,Zex6)).
three_dim_int(brick(A,B,C),Env,S) :- !,
        evaluate(A,A1,Env),
        evaluate(B,B1,Env),
        evaluate(C,C1,Env),
        get_coord(0,0,0,S,X1,Y1,Z1),
        get_coord(A1,0,0,S,X2,Y2,Z2),
        get_coord(0,B1,0,S,X3,Y3,Z3),
        get_coord(A1,B1,0,S,X4,Y4,Z4),
        get_coord(0,0,C1,S,X5,Y5,Z5),
        get_coord(A1,0,C1,S,X6,Y6,Z6),
        get_coord(0,B1,C1,S,X7,Y7,Z7),
        get_coord(A1,B1,C1,S,X8,Y8,Z8),
        my_window(gdlwindow_xy,V1,_),
        gmsend(V1,line(X1,Y1,X2,Y2)),
        gmsend(V1,line(X1,Y1,X3,Y3)),
        gmsend(V1,line(X2,Y2,X4,Y4)),
        gmsend(V1,line(X3,Y3,X4,Y4)),
        gmsend(V1,line(X1,Y1,X5,Y5)),
        gmsend(V1,line(X2,Y2,X6,Y6)),
        gmsend(V1,line(X3,Y3,X7,Y7)),
        gmsend(V1,line(X4,Y4,X8,Y8)),
        gmsend(V1,line(X5,Y5,X6,Y6)),
        gmsend(V1,line(X5,Y5,X7,Y7)),
        gmsend(V1,line(X6,Y6,X8,Y8)),
        gmsend(V1,line(X7,Y7,X8,Y8)),
        my_window(gdlwindow_xz,V2,_),
        gmsend(V2,line(X1,Z1,X2,Z2)),
        gmsend(V2,line(X1,Z1,X3,Z3)),
        gmsend(V2,line(X2,Z2,X4,Z4)),
        gmsend(V2,line(X3,Z3,X4,Z4)),
        gmsend(V2,line(X1,Z1,X5,Z5)),
        gmsend(V2,line(X2,Z2,X6,Z6)),
        gmsend(V2,line(X3,Z3,X7,Z7)),
        gmsend(V2,line(X4,Z4,X8,Z8)),
        gmsend(V2,line(X5,Z5,X6,Z6)),
        gmsend(V2,line(X5,Z5,X7,Z7)),
        gmsend(V2,line(X6,Z6,X8,Z8)),
        gmsend(V2,line(X7,Z7,X8,Z8)),
        my_window(gdlwindow_yz,V3,_),
        gmsend(V3,line(Y1,Z1,Y2,Z2)),
        gmsend(V3,line(Y1,Z1,Y3,Z3)),
        gmsend(V3,line(Y2,Z2,Y4,Z4)),
        gmsend(V3,line(Y3,Z3,Y4,Z4)),
        gmsend(V3,line(Y1,Z1,Y5,Z5)),
        gmsend(V3,line(Y2,Z2,Y6,Z6)),
        gmsend(V3,line(Y3,Z3,Y7,Z7)),
        gmsend(V3,line(Y4,Z4,Y8,Z8)),
```

```
      gmsend(V3,line(Y5,Z5,Y6,Z6)),
      gmsend(V3,line(Y5,Z5,Y7,Z7)),
      gmsend(V3,line(Y6,Z6,Y8,Z8)),
      gmsend(V3,line(Y7,Z7,Y8,Z8)).
three_dim_int(X,_,_) :- !,
      write('Internal error, the internal three '),
      write('dimensional statement '),nl,
      tab(2),write(X),nl,
      write(' is not recognized by the interpreter,'),
      write(' and skipped'),nl,nl.

%%% In the prism_-statement, there is a four bit mask represented by
%%% an integer, that we must separate into the four bits. We just use
%%% the three first bits.
decompress_mask([],[],[],[]).
decompress_mask([X|R],[C|R1],[B|R2],[A|R3]) :-
      (X > 7 -> X1 is X - 8 ; X1 = X),
      (X1 > 3 -> A = 1,X2 is X1 - 4 ; A = 0,X2 = X1),
      (X2 > 1 -> B = 1,X3 is X2 - 2 ; B = 0,X3 = X2),
      (X3 > 0 -> C = 1 ; C = 0),
      decompress_mask(R,R1,R2,R3).

%%% This procedure is used when a cylinder is to be plotted, when
%%% the points for the lines are to be found (i.e. the ellips'
%%% largest and smallest X- and Y-vals).
%%% It picks the largest and smallest X- and Y-vals from a list of
%%% vals, after the extremum value is found, i.e. the greatest length
%%% from the origo to the circle, which is done by
%%% find_extremum_points.
pick_the_extremum_points(N,Len,Xvals,Yvals,X1,Y1,X2,Y2) :-
      N1 is ((N + integer(Len / 2)) mod 36) + 1,
      pick(N,Xvals,X1),pick(N,Yvals,Y1),
      pick(N1,Xvals,X2),pick(N1,Yvals,Y2).

pick(1,[X|_],X) :- !.
pick(N,[_|R],X) :- N1 is N - 1,pick(N1,R,X).

%%% Finds the point that lies most far away from the origo of an
%%% ellips. The value is calculated with Pythagoras theorem
find_extremum_points(_,_,_,N,[],[],_,N) :- !.
find_extremum_points(X1,Y1,Sofar,Num,[X2|Xvals],[Y2|Yvals],N,AnswN) :-
      Tmp is sqrt(((X2 - X1) * (X2 - X1)) + ((Y2 - Y1) * (Y2 - Y1))),
      N1 is N + 1,
      (Tmp > Sofar ->
       find_extremum_points(X1,Y1,Tmp,N,Xvals,Yvals,N1,AnswN) ;
       find_extremum_points(X1,Y1,Sofar,Num,Xvals,Yvals,N1,AnswN)).

%%% plots all lines between the two polygons of a prisma
plot_prism_lines([],_,_,_).
plot_prism_lines([(X,Y)|R],Z,Env,State) :-
      two_dim_int(lin_(X,Y,0,X,Y,Z),Env,State),
      plot_prism_lines(R,Z,Env,State).

%%% plots all lines between the two polygons of a prism_-statement
plot_prism_lines_masked([],_,_,_,_).
plot_prism_lines_masked([(X,Y,_)|R],[M|R1],Z,Env,State) :-
      (1 is integer(M) ->
       two_dim_int(lin_(X,Y,0,X,Y,Z),Env,State) ;
       true),
      plot_prism_lines_masked(R,R1,Z,Env,State).

%%%---------------------------------
%%% plot_poly and plot_poly_with_mask plots a polygon.
```

```
%%% plot_poly makes use of ISP's polygon routine by first transforming
%%% all local points to global values, and then plot a polygon in each
%%% view.
plot_poly(H,X,Y,E) :- plot_poly(H,X,Y,E,[],[],[]).

plot_poly(_,[],[],_,Xvals,Yvals,Zvals) :-
      my_window(gdlwindow_xy,V1,_),gmsend(V1,polygon(Xvals,Yvals)),
      my_window(gdlwindow_xz,V2,_),gmsend(V2,polygon(Xvals,Zvals)),
      my_window(gdlwindow_yz,V3,_),gmsend(V3,polygon(Yvals,Zvals)).
plot_poly(H,[X|R],[Y|R1],State,Xsofar,Ysofar,Zsofar) :-
      get_coord(X,Y,H,State,Xval,Yval,Zval),
      plot_poly(H,R,R1,State,[Xval|Xsofar],
                [Yval|Ysofar],[Zval|Zsofar]).

%%% plot_poly_with_mask plots a polygon, where some of the lines can
%%% be invisible. Therefore we cannot use ISP's polygon routine, so
%%% each line is plotted individually.
plot_poly_with_mask(H,[M1|Masks],[X1|R1],[Y1|R2],State,Env) :-
      plot_poly_with_mask1(H,M1,X1,Y1,State,Env,Masks,R1,R2,X1,Y1).

plot_poly_with_mask1(H,Mask,X,Y,S,Env,[],[],[],X1,Y1) :- !,
      (1 is integer(Mask) ->
       two_dim_int(lin_(X,Y,H,X1,Y1,H),Env,S) ;
       true).
plot_poly_with_mask1(H,Mask,X,Y,S,Env,[M|R],[X1|R1],[Y1|R2],Xn,Yn) :-
      (1 is integer(Mask) ->
       two_dim_int(lin_(X,Y,H,X1,Y1,H),Env,S) ;
       true),
      plot_poly_with_mask1(H,M,X1,Y1,S,Env,R,R1,R2,Xn,Yn).

%%% Evaluates a list of coordinates, where each coordinate is
%%% specified by an X and a Y coordinate. The expressions are just
%%% evaluated, no transforming are performed.
evaluate_coord2_list([],[],[],_).
evaluate_coord2_list([(X,Y)|R],[X1|R1],[Y1|R2],Env) :-
      evaluate(X,X1,Env),evaluate(Y,Y1,Env),
      evaluate_coord2_list(R,R1,R2,Env).

%%% The same as evaluate_coord, except that the coordinates also has a
%%% mask.
evaluate_coord2_list_mask([],[],[],[],_).
evaluate_coord2_list_mask([(X,Y,Mask)|R],[X1|R1],[Y1|R2],
                          [Mask1|R3],Env) :-
      evaluate(X,X1,Env),evaluate(Y,Y1,Env),evaluate(Mask,Mask1,Env),
      evaluate_coord2_list_mask(R,R1,R2,R3,Env).

%%%---------------------------------
%%% General routine for evaluating functional expressions, with the
%%% syntax from the file gdlparser.pl. The first argument is the
%%% expression, The second argument is the value of that expression,
%%% and the third argument is the current variable environment.
%%% evaluate uses ISP's math library when possible.
evaluate(X,X,_) :- number(X),!.
evaluate(var(X),Y,Env) :- !,lookup(X,Y,Env).
evaluate(constant(pi),X,_) :- !,
      asin(1,Tmp),X is 2 * Tmp.
evaluate(function(fraction,X),Answ,Env) :- !,
      evaluate(X,X1,Env),
      Z is X1 - integer(X1),
      roundoff(Z,6,Answ).
evaluate(function(int,X),Y,Env) :- !,
      evaluate(X,X1,Env),
```

```
      aint(X1,Y).
evaluate(function(sgn,X),Y,Env) :- !,
      evaluate(X,X1,Env),
      (X1 > 0 -> Y = 1 ;
       (X1 < 0 -> Y = -1 ; Y = 0)).
evaluate(function(acos,X),Y,Env) :- !,
      evaluate(X,X1,Env),
      acos(X1,Y1),
      Y is Y1 * (360 / 6.2831852).
evaluate(function(asin,X),Y,Env) :- !,
      evaluate(X,X1,Env),
      asin(X1,Y1),
      Y is Y1 * (360 / 6.2831852).
evaluate(function(atan,X),Y,Env) :- !,
      evaluate(X,X1,Env),
      atan(X1,Y1),
      Y is Y1 * (360 / 6.2831852).
evaluate(function(cos,X),Y,Env) :- !,
      evaluate(X,X1,Env),
      X2 is X1 * (6.2831852 / 360),
      cos(X2,Y).
evaluate(function(sin,X),Y,Env) :- !,
      evaluate(X,X1,Env),
      X2 is X1 * (6.2831852 / 360),
      sin(X2,Y).
evaluate(function(tan,X),Y,Env) :- !,
      evaluate(X,X1,Env),
      X2 is X1 * (6.2831852 / 360),
      tan(X2,Y).
evaluate(function(exp,X),Y,Env) :- !,
      evaluate(X,X1,Env),
      exp(X1,Y).
evaluate(function(lgt,X),Y,Env) :-
      evaluate(X,X1,Env),
      log10(X1,Y).
evaluate(function(log,X),Y,Env) :-
      evaluate(X,X1,Env),
      log(X1,Y).
evaluate(operator(_,=,A,B),Y,Env) :- !,
      evaluate(A,A1,Env),evaluate(B,B1,Env),
      (float(A1) =:= float(B1) -> Y = 1 ; Y = 0).
evaluate(operator(_,<,A,B),Y,Env) :- !,
      evaluate(A,A1,Env),evaluate(B,B1,Env),
      (float(A1) < float(B1) -> Y = 1 ; Y = 0).
evaluate(operator(_,>,A,B),Y,Env) :- !,
      evaluate(A,A1,Env),evaluate(B,B1,Env),
      (float(A1) > float(B1) -> Y = 1 ; Y = 0).
evaluate(operator(_,<=,A,B),Y,Env) :- !,
      evaluate(A,A1,Env),evaluate(B,B1,Env),
      (float(A1) =< float(B1) -> Y = 1 ; Y = 0).
evaluate(operator(_,>=,A,B),Y,Env) :- !,
      evaluate(A,A1,Env),evaluate(B,B1,Env),
      (float(A1) >= float(B1) -> Y = 1 ; Y = 0).
evaluate(operator(_,<>,A,B),Y,Env) :- !,
      evaluate(A,A1,Env),evaluate(B,B1,Env),
      ((float(A1) =\= float(B1)) -> Y = 1 ; Y = 0).
evaluate(function(not,X),Y,Env) :- !,
      evaluate(X,X1,Env),
      (float(X1) =:= 0.0 -> Y = 1 ; Y = 0).
evaluate(operator(_,and,X,Y),Z,Env) :- !,
      evaluate(X,X1,Env),
      evaluate(Y,Y1,Env),
      (round(X1) =\= 0,round(Y1) =\= 0 -> Z = 1 ; Z = 0).
```

```
evaluate(operator(_,or,X,Y),Z,Env) :- !,
      evaluate(X,X1,Env),
      evaluate(Y,Y1,Env),
      ((round(X1) =\= 0 ; round(Y1) =\= 0) -> Z = 1 ; Z = 0).
evaluate(operator(_,xor,X,Y),Z,Env) :- !,
      evaluate(X,X1,Env),
      evaluate(Y,Y1,Env),
      ((round(X1) =\= 0, round(Y1) =\= 0 ;
        round(X1) =:= 0, round(Y1) =:= 0) ->
       Z = 0 ; Z = 1).
evaluate(function(F,X),Z,Env) :- !,evaluate(X,X1,Env),
      Z1=..[F,X1],Z is Z1.
evaluate(operator(_,F,X,Y),Z,Env) :- !,
      evaluate(X,X1,Env),evaluate(Y,Y1,Env),Z1=..[F,X1,Y1], Z is Z1.
evaluate(function(F,X),_,_) :- !,
      write('Error: '),write(F),write('('),write(X),write(')'),
      write(' is not a proper expression'),nl,abort.
evaluate(constant(X),0,_) :- !,
      write('Error, constant not implemented: '),write(X),
      write(' Replacing the constant with the value 0'),nl.
evaluate(operator(_,F,X,Y),_,_) :- !,
      write('Error: '),write(X),write(' '),write(F),write(' '),
      write(Y), write(' is not a proper expression'),nl,abort.

%%%---------------------------------
%%% Some general routines

%%% Calculating a point's local coordinates into global coordinates,
%%% scale it into "X-coordinates" for the X window system.
get_coord(X,Y,Z,[M|_],Xval,Yval,Zval) :-
      vectormult([X,Y,Z,1],M,[X1,Y1,Z1,Sc]),
      Xval is integer(X1 * 250 / Sc),
      Yval is integer(Y1 * 250 / Sc),
      Zval is integer(Z1 * 250 / Sc).

%%% Binds Var to Val, i.e. if Var exists, its old value is replaced
%%% by the new one, and if Var does not exist, it is introduced
%%% and bound to Val.
make_env(Var,Val,[(Var,_)|R],[(Var,Val)|R]) :- !.
make_env(Var,Val,[],[(Var,Val)]) :- !.
make_env(Var,Val,[F|R],[F|R1]) :- make_env(Var,Val,R,R1).

%%% Finds var in the current environment. If it is not there,
%%% the default value 0.00 is returned.
lookup(Var,Val,[(Var,Val)|_]) :- !.
lookup(_,0.00,[]) :- !.
lookup(Var,Val,[_|R]) :- lookup(Var,Val,R).

%%% Reads symbols to an EOL is reached
read_to_eol(X) :- get_letters(X1),name(X,X1).

get_letters(X) :-
      get0(Z),(Z = 10 -> X = [] ; get_letters(R),X = [Z|R]).

%%% General routine for multiplying two matrixes. A matrix is
%%% represented as a list of rows, which are themselves lists, for
%%% example the matrix
%%%    3  4
%%%    6  7
%%% is represented by [[3,4],[6,7]].
%%% We have only implemented multiplication, since that is the only
%%% operation we need.
matrixmult([],_,[]).
```

```
matrixmult([F|R],L,[F1|R1]) :-          % Each row are calculated
      vectormult(F,L,F1),
      matrixmult(R,L,R1).

%%% This routine multiplies a vector to a matrix
vectormult(_,[[]|_],[]) :- !.
vectormult(V,L,[Answ|R]) :-
      matrixmult1(V,L,0,L1,Answ),       % Each column is traversed
      vectormult(V,L1,R).

%%% We traverse the first column of the second argument, adding
%%% each value to the third argument, and returning the sum in the
%%% fifth argument. The fourth argument is the third argument without
%%% the first column.
matrixmult1(_,[],Answ,[],Answ).
matrixmult1([F|R],[[F1|R1]|R2],Sofar,[R1|R3],Answ) :-
      Tmp is F * F1 + Sofar,
      matrixmult1(R,R2,Tmp,R3,Answ).
```